Search Techniques

Searching for a (shortest / least cost) path to goal state(s).

Search through the state space.

We will consider search techniques that use an explicit search tree that is generated by the initial state + successor function.

initialize (initial node)
Loop
choose a node for expansion according to strategy
goal node? → done
expand node with successor function

Tree-search algorithms

Basic idea:

- **simulated exploration of state space by generating successors of already-explored states (a.k.a. ~ expanding states)**

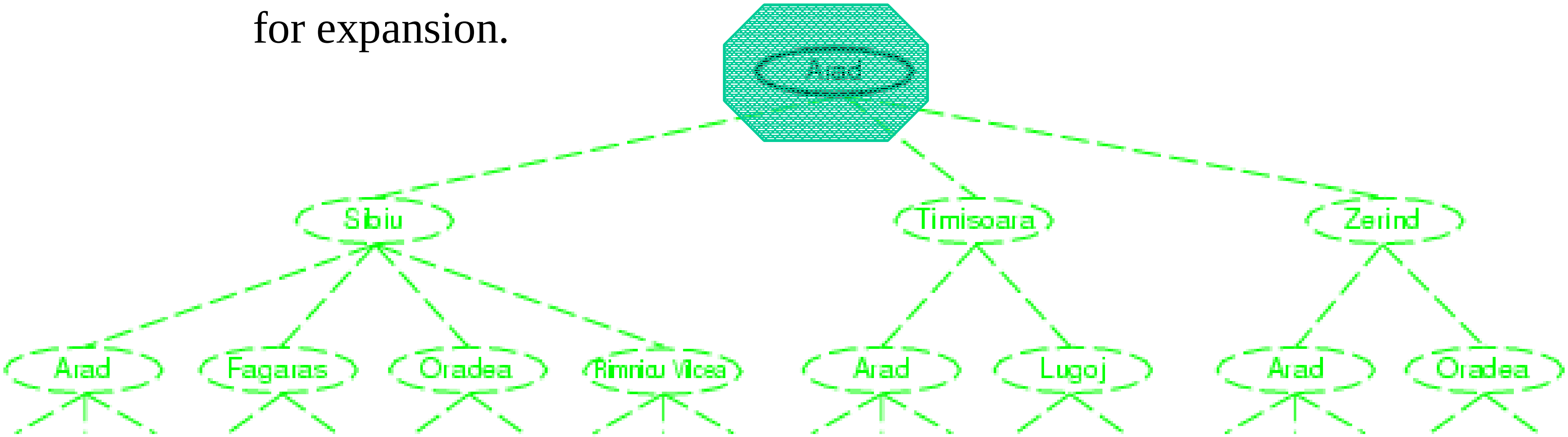
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal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

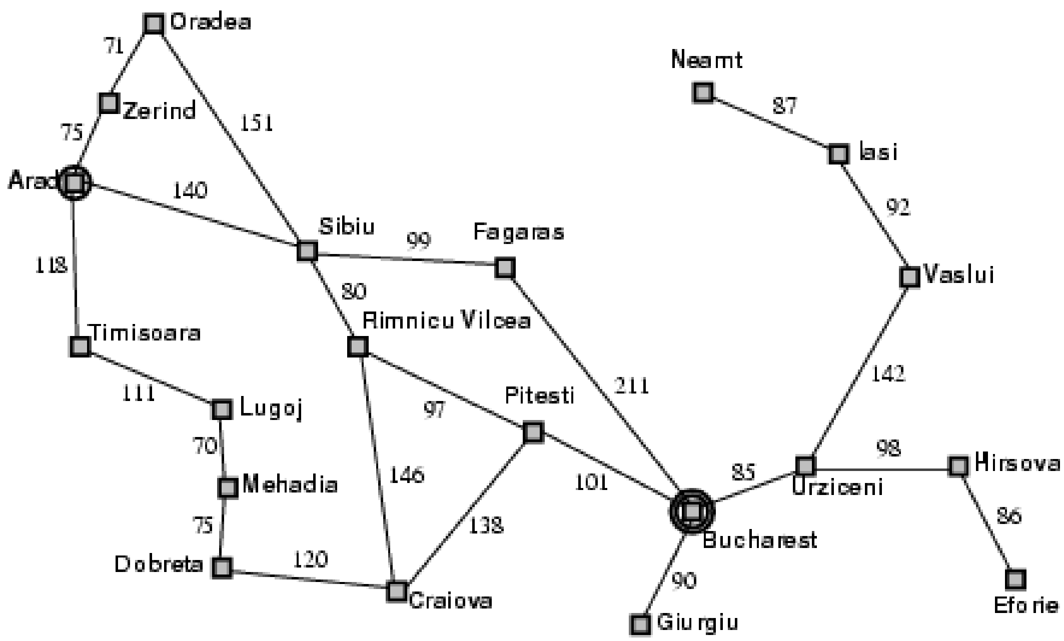
Fig. 3.7 R&N, p. 77

Note: 1) Here we only check a node for possibly being a goal state, after we select the node for expansion.
2) A "node" is a data structure containing state + additional info (parent node, etc.

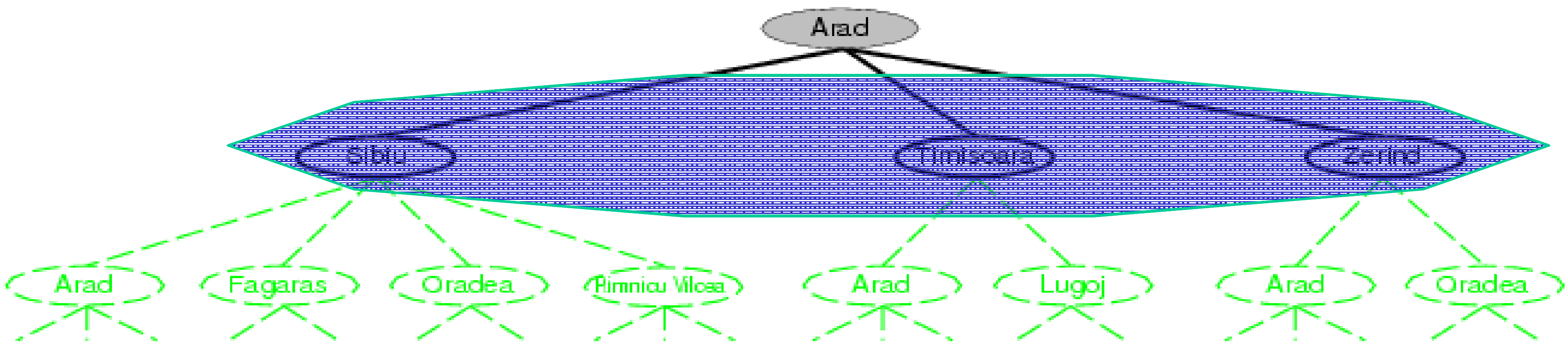
Tree search example

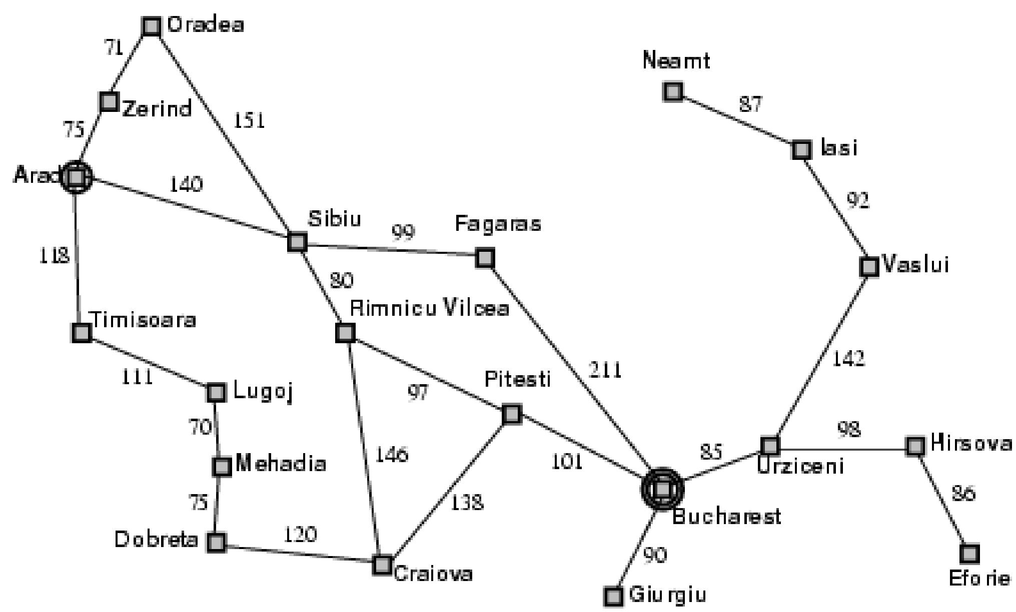
Node selected
for expansion.





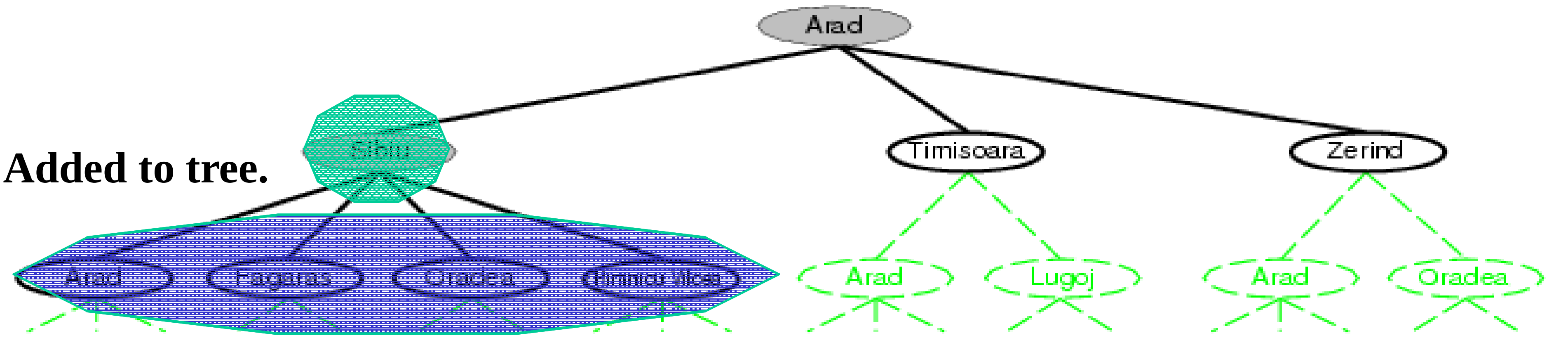
Nodes added to tree.





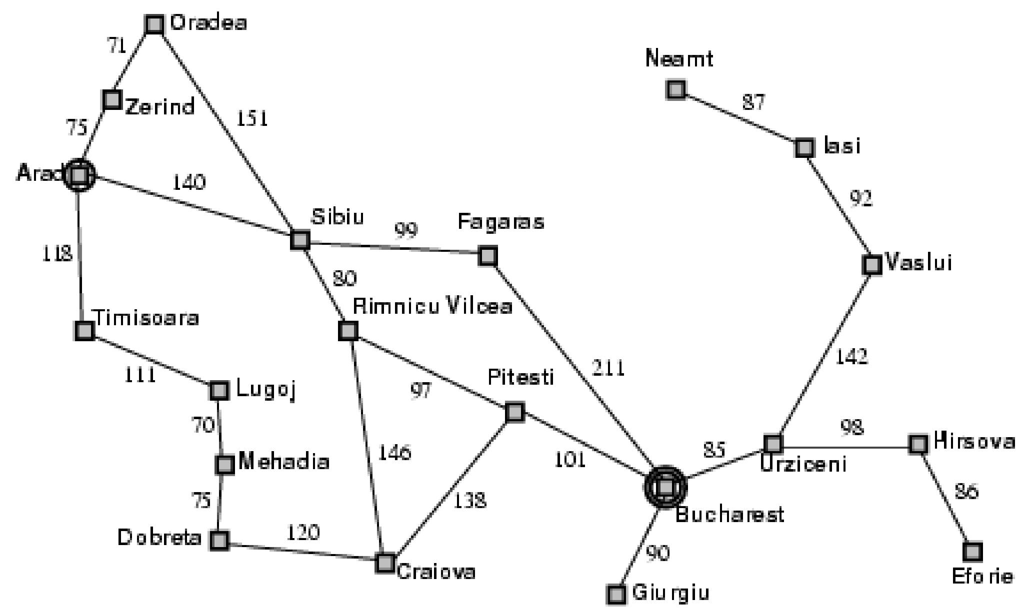
Selected for expansion.

Added to tree.



Note: Arad added (again) to tree!
(reachable from Sibiu)

Not necessarily a problem, but in Graph-Search, we will avoid this by maintaining an “explored” list.



Graph-search

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  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
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Note:

1) Uses "explored" set to avoid visiting already explored states.

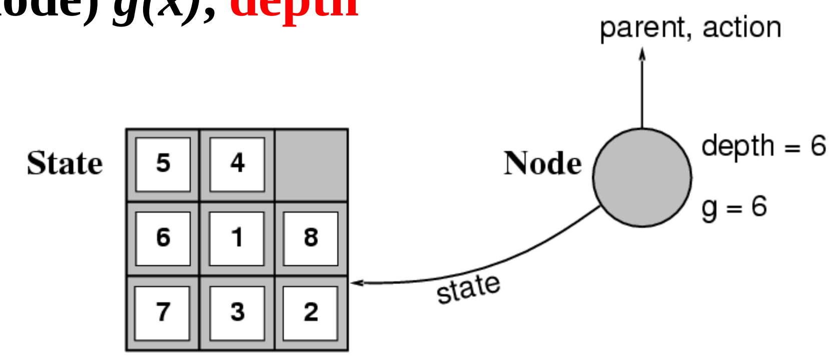
2) Uses "frontier" set to store states that remain to be explored and expanded.

3) *However, with eg uniform cost search, we need to make a special check when node (i.e. state) is on frontier. Details later.*

Implementation: states vs. nodes

A state is a --- representation of --- a physical configuration.

A node is a data structure constituting part of a search tree includes state, tree parent node, action (applied to parent), path cost (initial state to node) *g(x)*, depth



The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Fringe is the collection of nodes that have been generated but not (yet) expanded. Each node of the fringe is a leaf node.

Implementation: general tree search

```
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
```

Search strategies

A search 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**
- **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 ∞)**
-

Uninformed search strategies

Uninformed (blind) search 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**
- **Bidirectional search**
-

Key issue: type of queue used for the fringe of the search tree (collection of tree nodes that have been generated but not yet expanded)

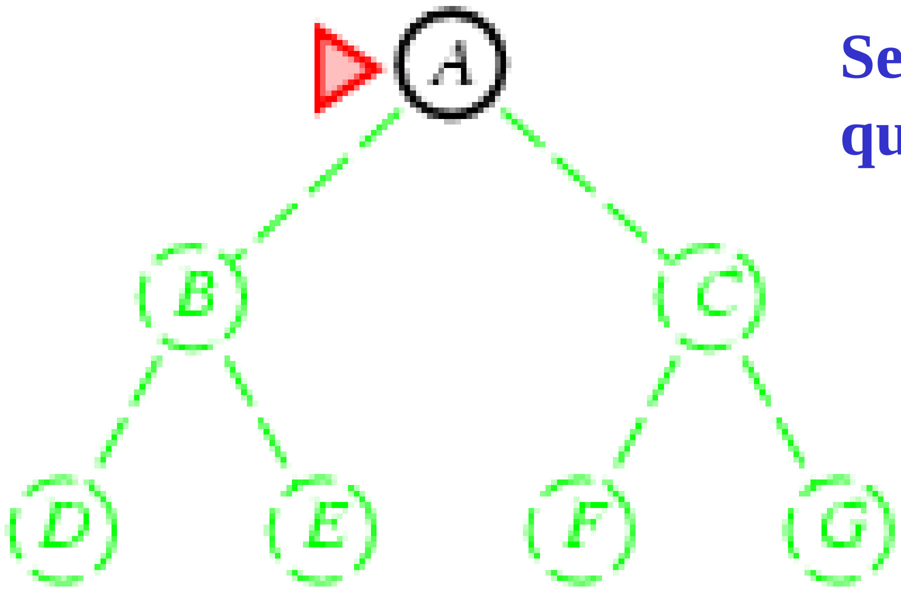
Breadth-first search

Expand shallowest unexpanded node.

Implementation:

- ***fringe* is a FIFO queue, i.e., new nodes go at end (First In First Out queue.)**

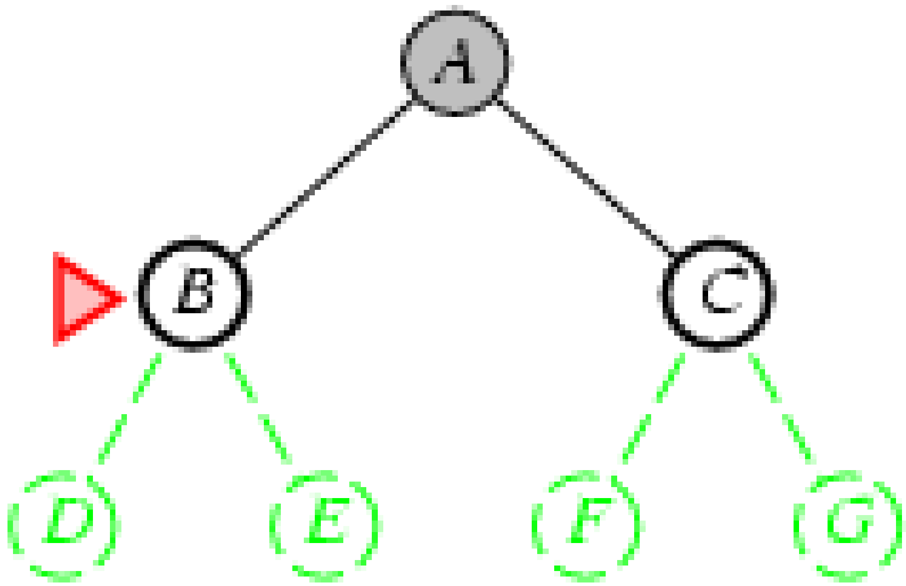
Fringe queue: <A>



Select A from queue and expand.

Gives <B, C>

Queue: <B, C>

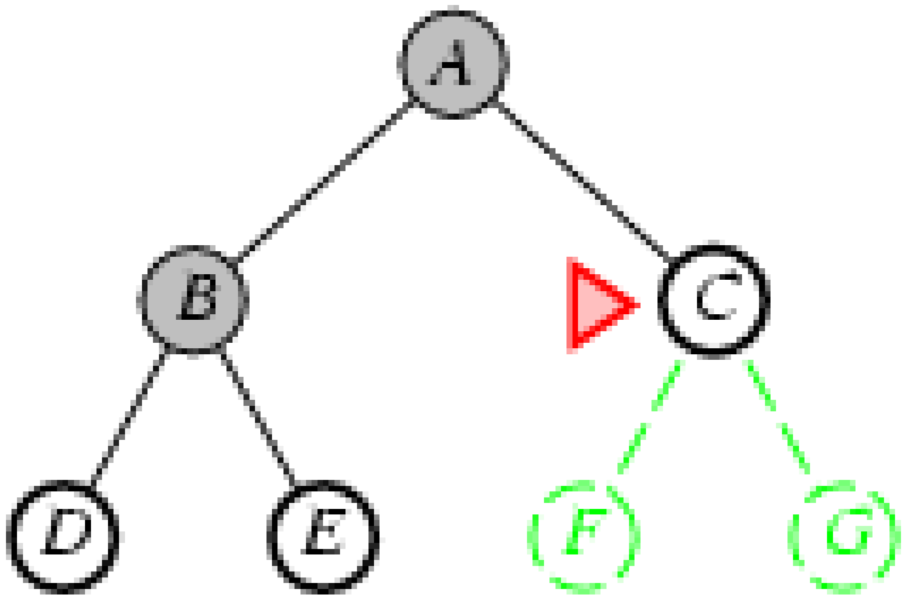


Select B from front, and expand.

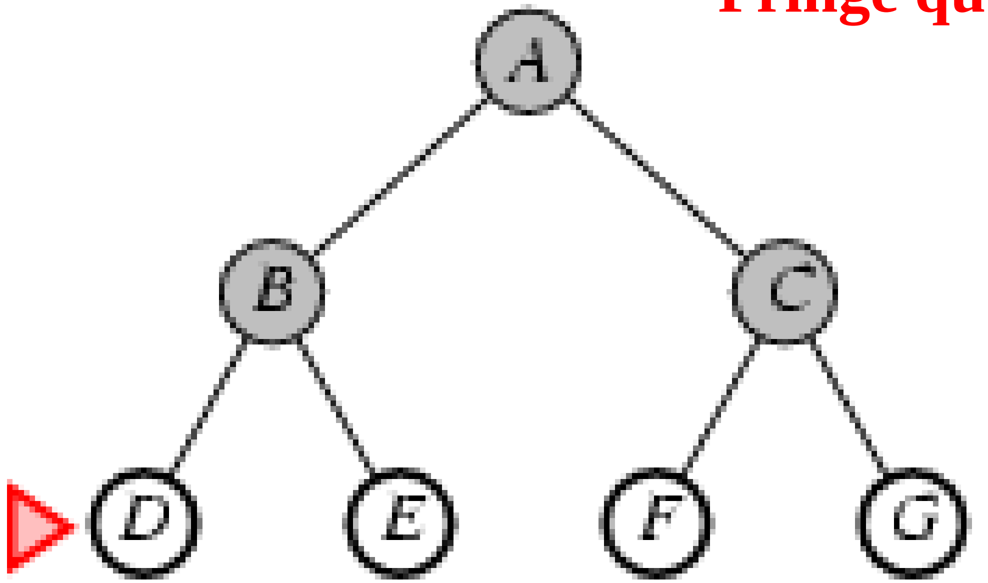
Put children at the end.

Gives
<C, D, E>

Fringe queue: <C, D, E>



Fringe queue: <D, E, F, G>



Assuming no further children, queue becomes <E, F, G>, <F, G>, <G>, <>. Each time node checked for goal state.

Properties of breadth-first search

Note: check for goal only when node is expanded.

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; needed also to reconstruct soln. path)

Optimal soln. found?

Yes (if all step costs are identical)

Space is the bigger problem (more than time)

b: maximum branching factor of the search tree
d: depth of the least-cost solution

Uniform-cost search

Expand least-cost (of path to) unexpanded node

(e.g. useful for finding shortest path on map)

Implementation:

- ***fringe* = queue *ordered by path cost***
- **g – cost of reaching a node**

Complete? Yes, if step cost $\geq \varepsilon$ **(>0)**

Time? # of nodes with $g \leq$ cost of optimal solution (C*), $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Optimal? Yes – nodes expanded in increasing order of $g(n)$

Note: Some subtleties (e.g. checking for goal state).

See p 84 R&N. Also, next slide.

Uniform-cost search

Two subtleties: (bottom p. 83 Norvig)

1) **Do goal state test, only when a node is selected for expansion.**
   (Reason: Bucharest may occur on frontier with a longer than optimal path. It won't be selected for expansion yet. Other nodes will be expanded first, leading us to uncover a shorter path to Bucharest. See also point 2).

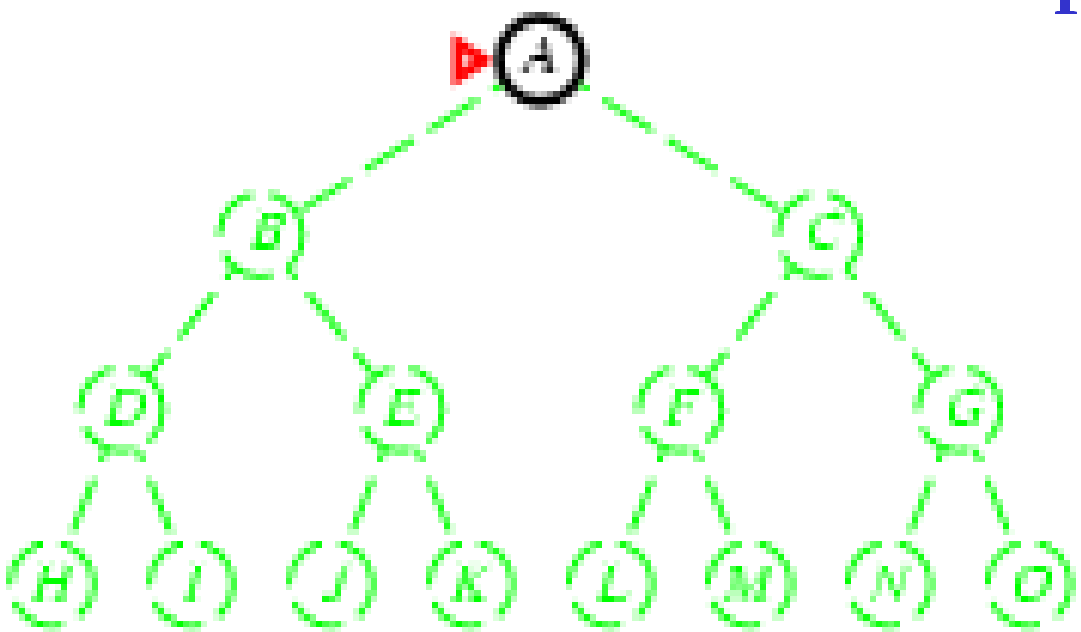
2) **Graph-search alg. says "don't add child node to frontier if already on explored list or already on frontier." BUT, child may give a shorter path to a state already on frontier. Then, we need to modify the existing node on frontier with the shorter path. See fig. 3.14 (else-if part).**

Depth-first search

"Expand deepest unexpanded node"

Implementation:

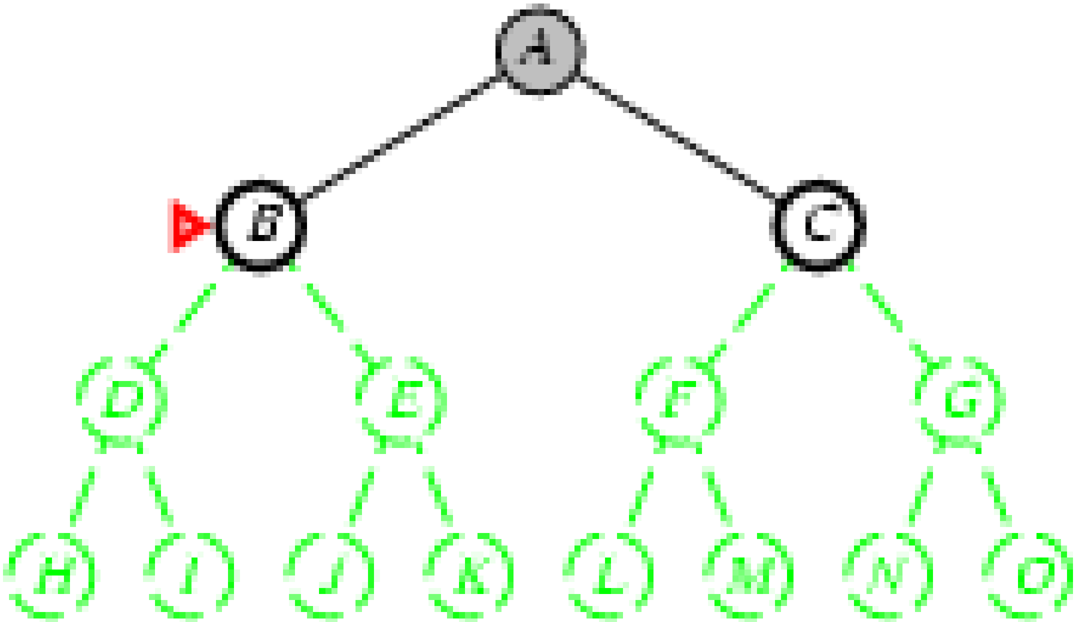
- *fringe* = LIFO queue, i.e., put successors at front ("push on stack")
  Last In First Out



Fringe stack:
A

Expanding A,
gives stack:
B
C

So, B next.



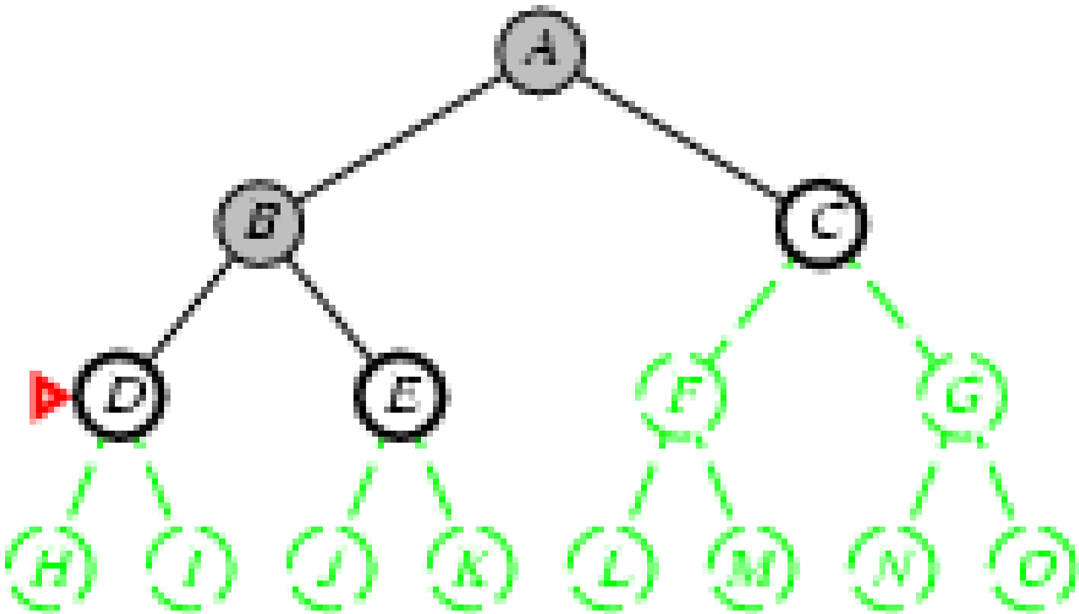
Expanding B, gives stack:

D

E

C

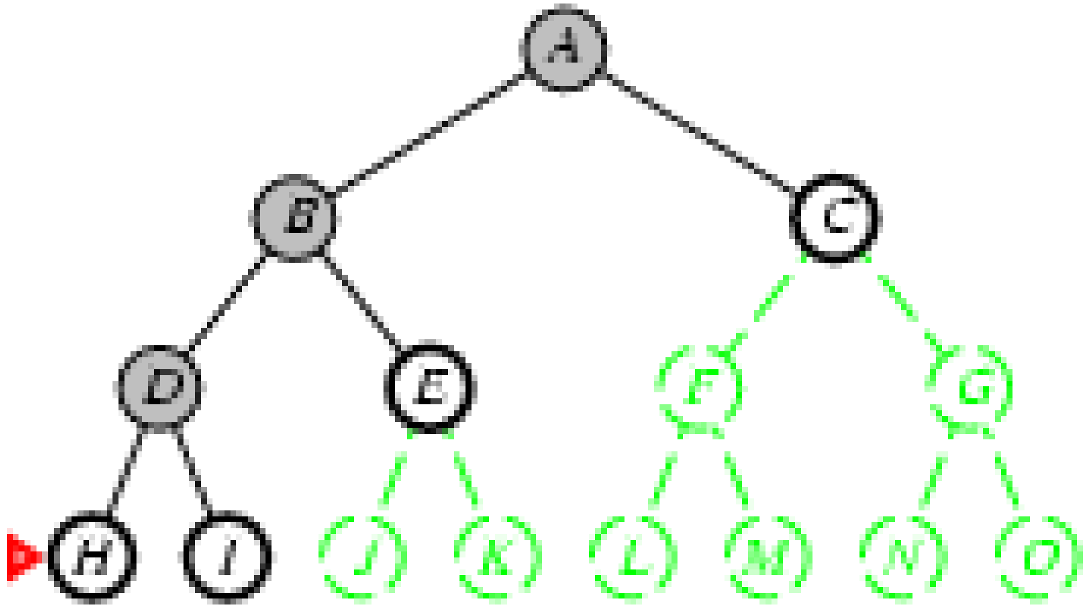
So, D next.

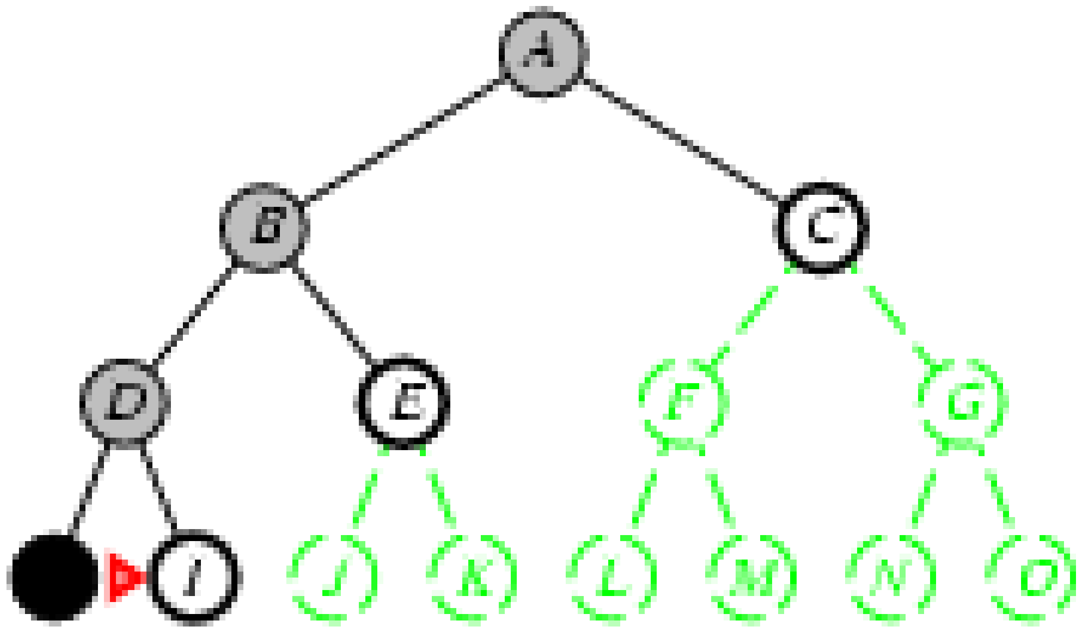


Expanding D,
gives stack:

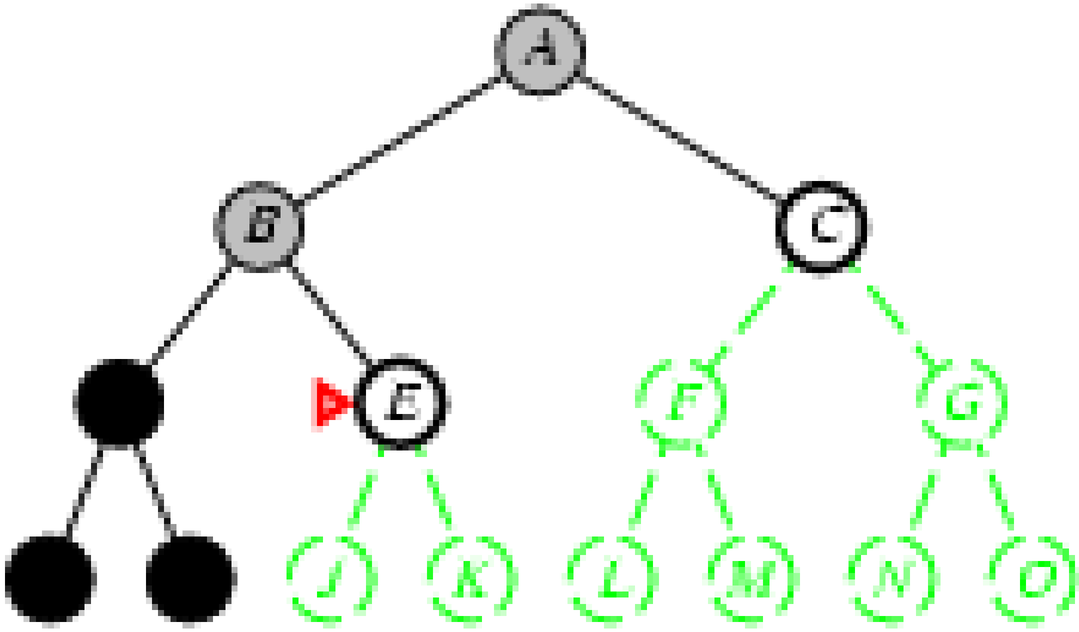
H
I
E
C

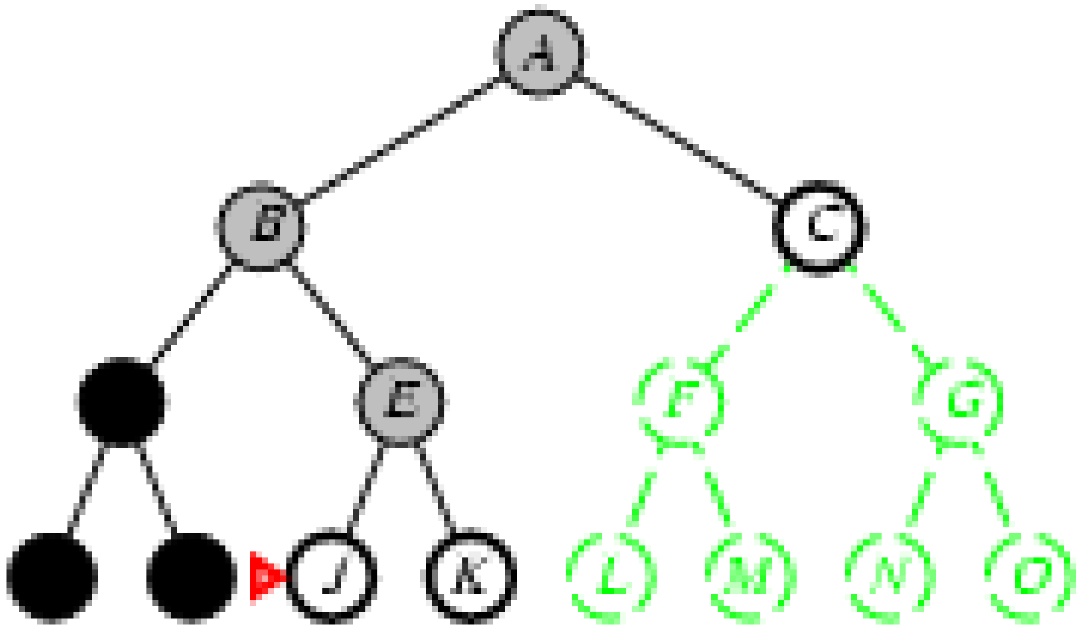
So, H next.
etc.

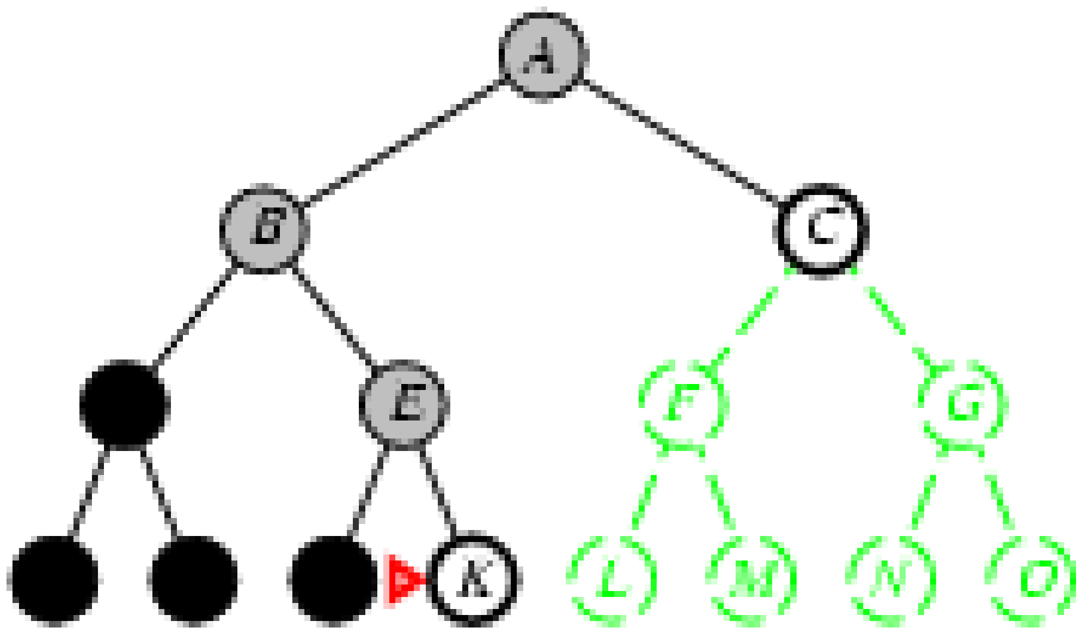

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

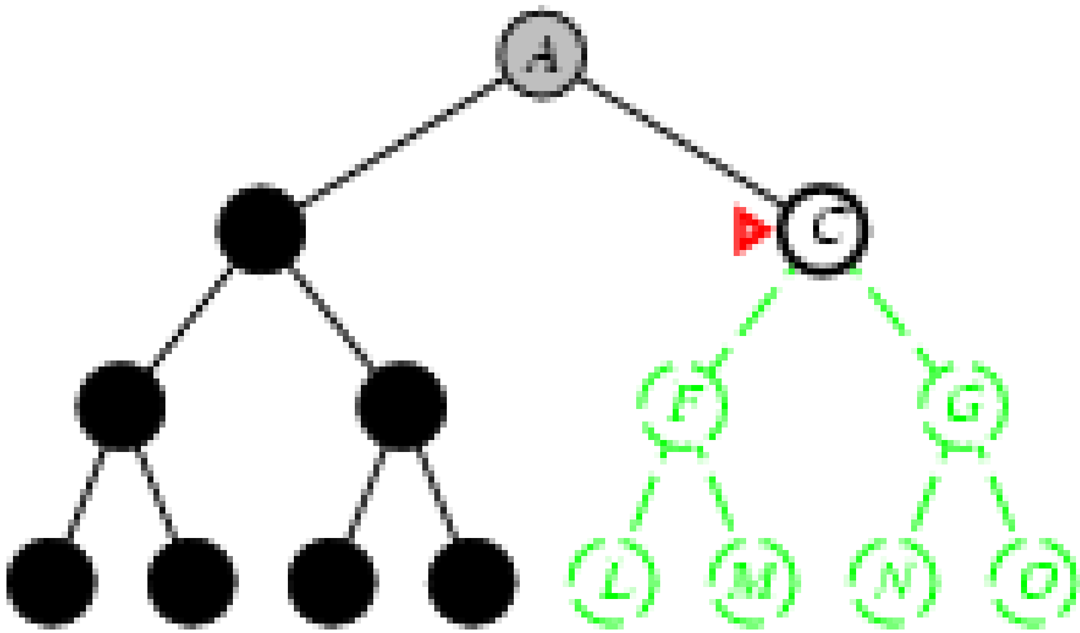


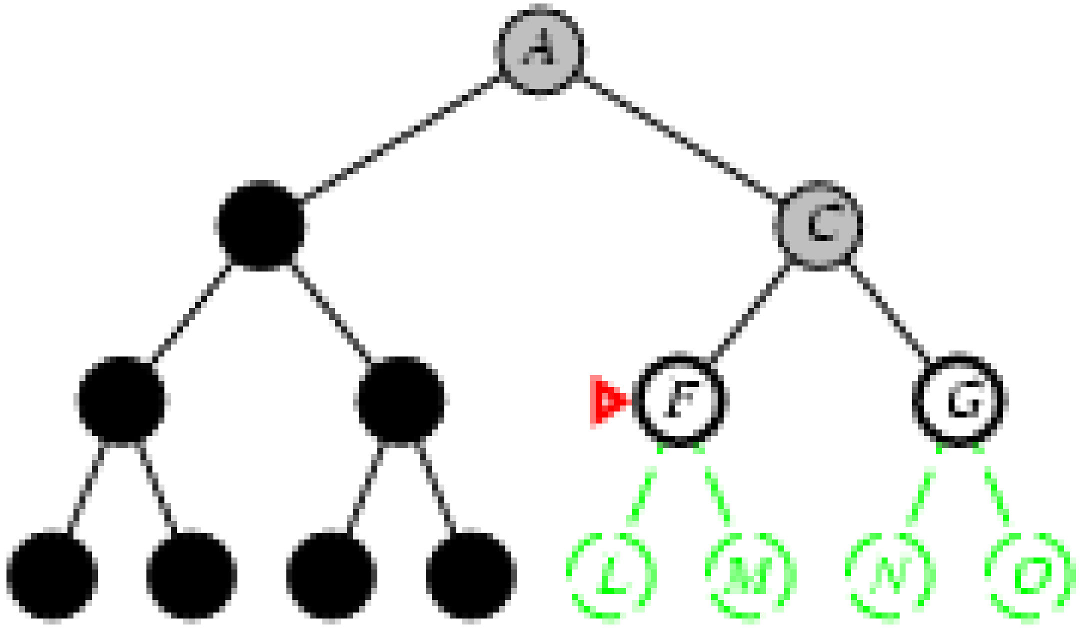
A
B
C
D
E
F
G
I
J
K
L
M
N
O


A
B
C
E
F
G
J
K
L
M
N
O

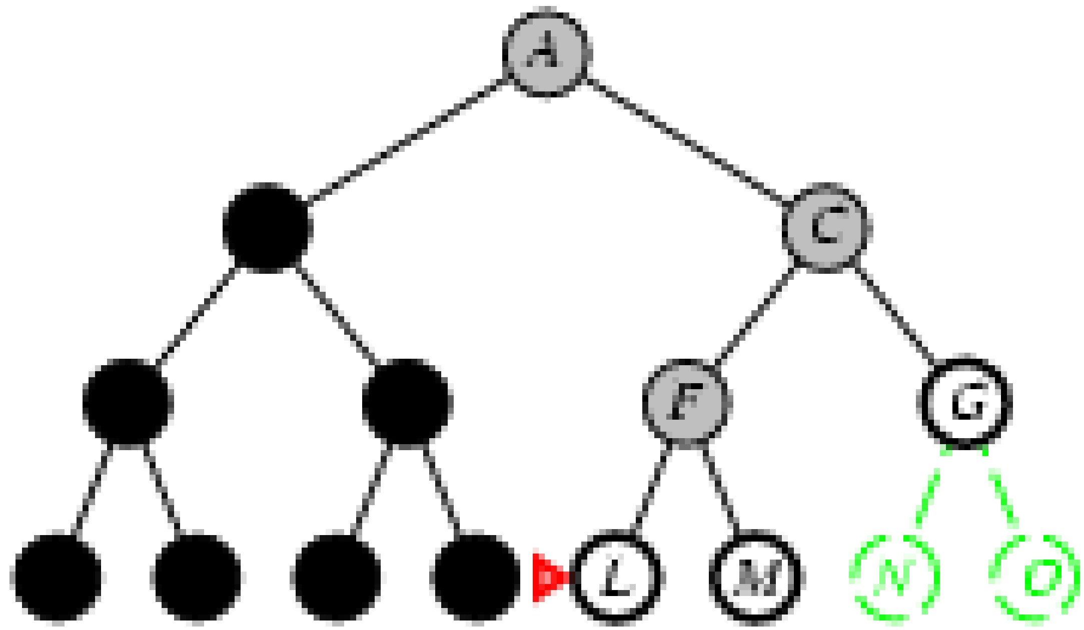

A
B
C
E
F
G
J
K
L
M
N
O


A
B
C
E
F
G
K
L
M
N
O

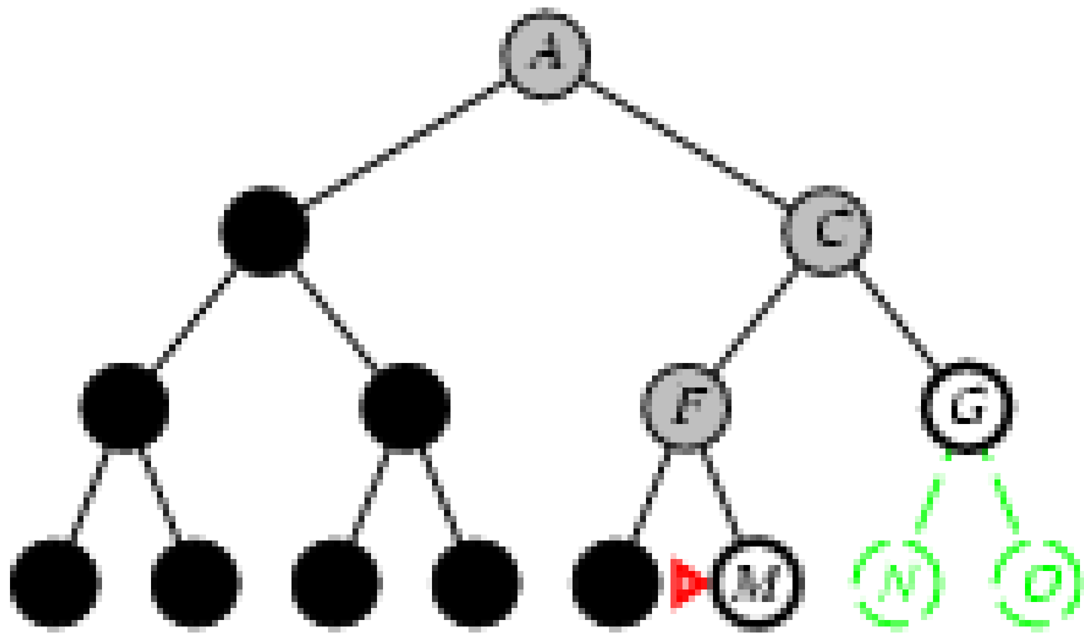

A
C
F
G
L
M
N
O



A
C
F
G
L
M
N
O



A
C
F
G
L
M
N
O


A
C
F
G
M
N
O

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)*: bad 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!**

Note: Can also reconstruct soln. path from single stored branch.

Guarantee that opt. soln. is found? No

***b:* max. branching factor of the search tree**
***d:* depth of the shallowest (least-cost) soln.**
m: maximum depth of state space

Note: In "backtrack search" only one successor is generated
→ only O(m) memory is needed; also successor is modification of the current state, but we have to be able to undo each modification. More when we talk about Constraint Satisfaction Problems (CSP).

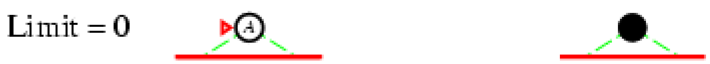
[here]

Iterative deepening search

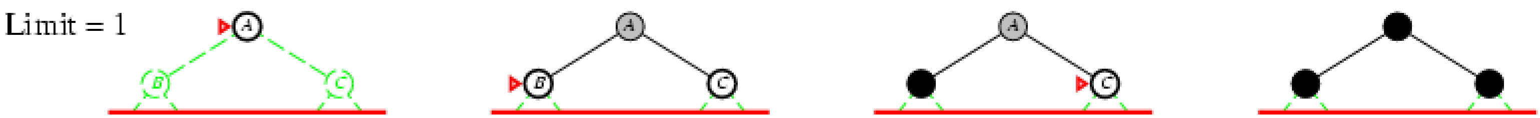
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
     inputs: problem, a problem

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

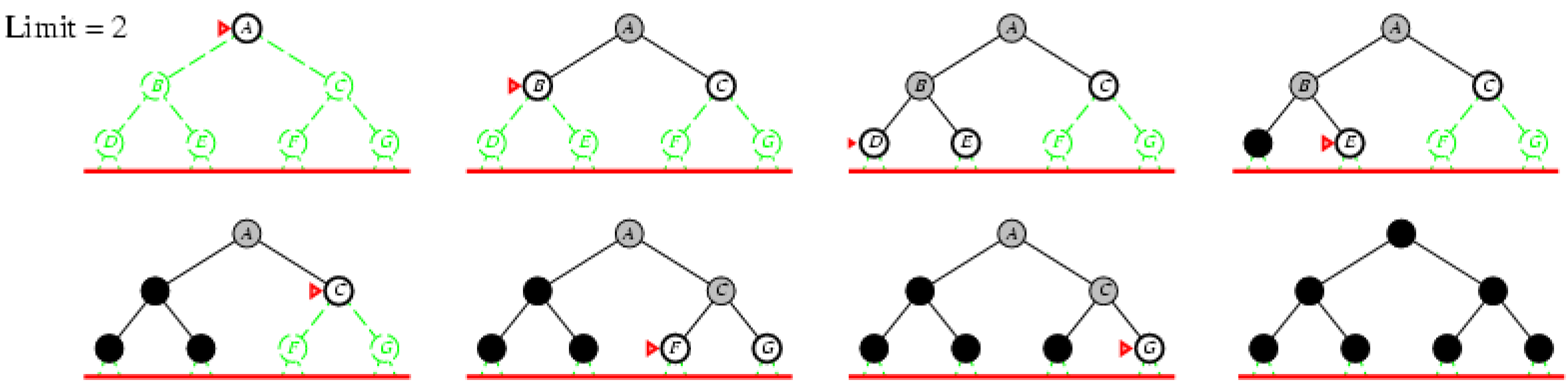
Iterative deepening search *l* =0



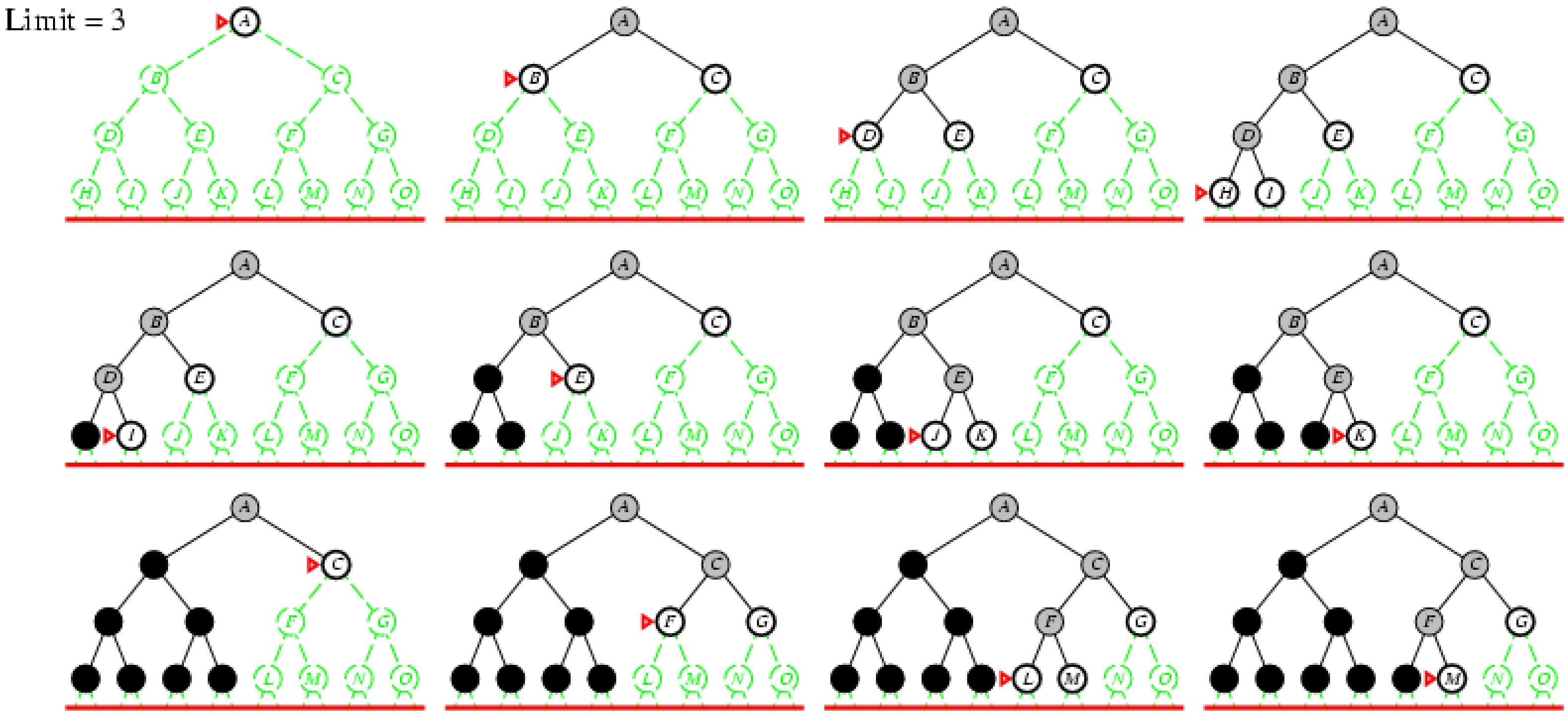
Iterative deepening search l =1



Iterative deepening search l =2



Iterative deepening search *l* =3



Why would one do that?

Combine good memory requirements of depth-first with the completeness of breadth-first when branching factor is finite and is optimal when the path cost is a non-decreasing function of the depth of the node.

Idea was a breakthrough in game playing. All game tree search uses iterative deepening nowadays. What's the added advantage in games?

"Anytime" nature.

Iterative deepening search

Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

Looks quite wasteful, is it?

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

Nodes generated in a breadth-first search with branching factor *b*:

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

For *b = 10, d = 5,*

- $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$
-
- $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$ ☺
-

Iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Properties of iterative deepening search

Complete? **Yes**

(*b* finite)

Time? $d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

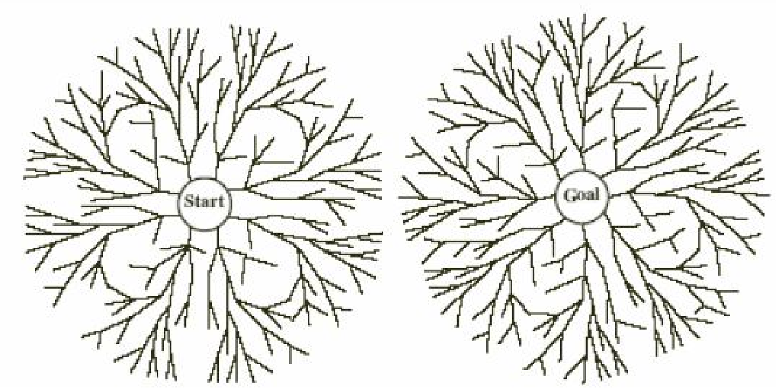
Space? $O(bd)$

Optimal? **Yes, if step costs identical**

Bidirectional Search

- **Simultaneously:**
  - **Search forward from start**
  - **Search backward from the goal**

  Stop when the two searches meet.



- **If branching factor = b in each direction, with solution at depth d**
  ➔ **only $O(2\ b^{d/2}) = O(2\ b^{d/2})$**

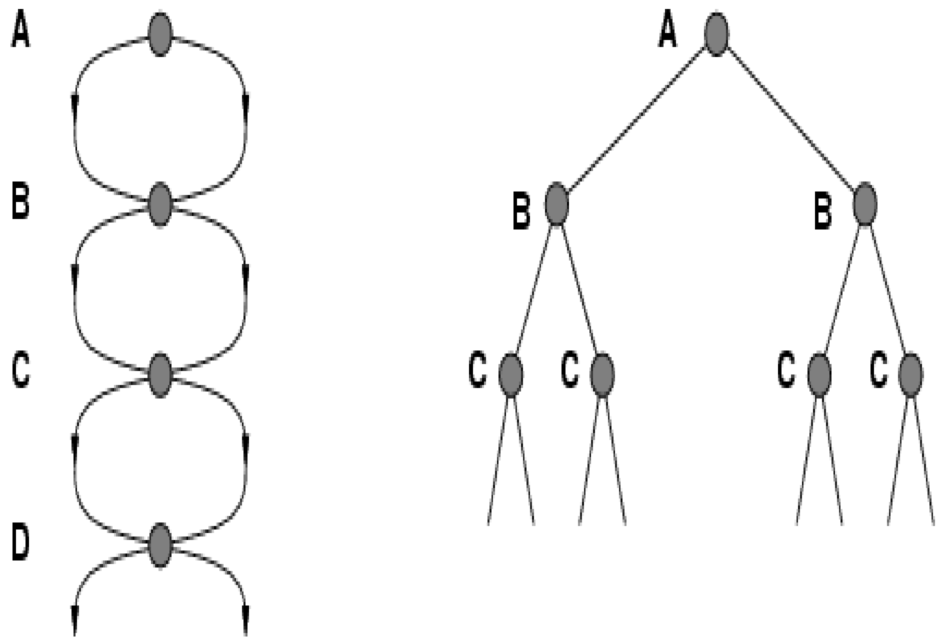
- **Checking a node for membership in the other search tree can be done in constant time (hash table)**

- **Key limitations:**
  Space $O(b^{d/2})$
  Also, how to search backwards can be an issue (e.g., in Chess)? What's tricky?
  Problem: lots of states satisfy the goal; don't know which one is relevant.

Aside: The predecessor of a node should be easily computable (i.e., actions are easily reversible).

Failure to detect repeated states can turn linear problem into an exponential one!

Repeated states



Don't return to parent node

Don't generate successor = node's parent

Don't allow cycles

Don't revisit state

Keep every visited state in memory!
$O(b^d)$ (can be expensive)

Problems in which actions are reversible (e.g., routing problems or sliding-blocks puzzle). Also, in eg Chess; uses hash tables to check for repeated states. Huge tables 100M+ size but very useful.

See Tree-Search vs. Graph-Search in Fig. 3.7 R&N. But need to be careful to maintain (path) optimality and completeness.

Summary: General, uninformed search

Original search ideas in AI where inspired by studies of human problem solving in, eg, puzzles, math, and games, but a great many AI tasks now require some form of search (e.g. find optimal agent strategy; active learning; constraint reasoning; NP-complete problems require search).

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.

Avoid repeating states / cycles.