Search Strategies

Reading: Russell's Chapter 3

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/expanded
- space complexity: maximum number of nodes in memory
- optimality: does it always find a least-cost solution?

Time and space complexity are measured in terms of

- *b:* maximum branching factor of the search tree
- *d:* depth of the least-cost solution
- *m*: maximum depth of the state space (may be ∞)

Uninformed search strategies

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

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

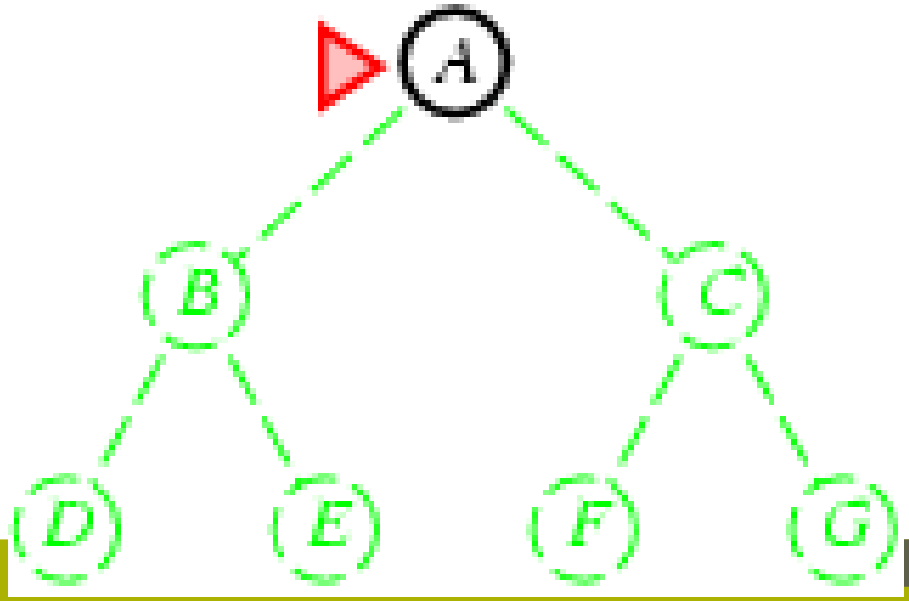
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

– *fringe* is a FIFO queue, i.e., new successors go at end

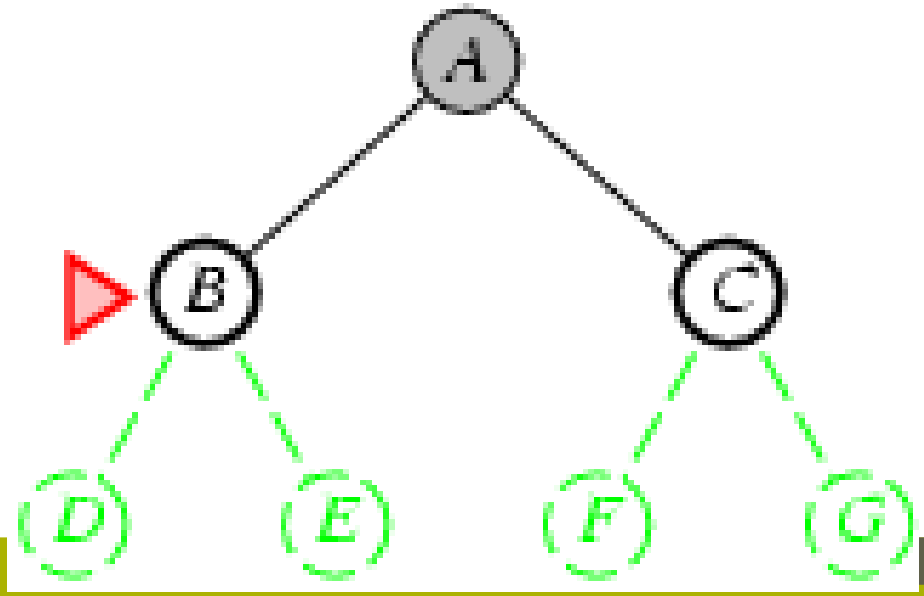


Breadth-first search

Expand shallowest unexpanded node

Implementation:

– *fringe* is a FIFO queue, i.e., new successors go at end

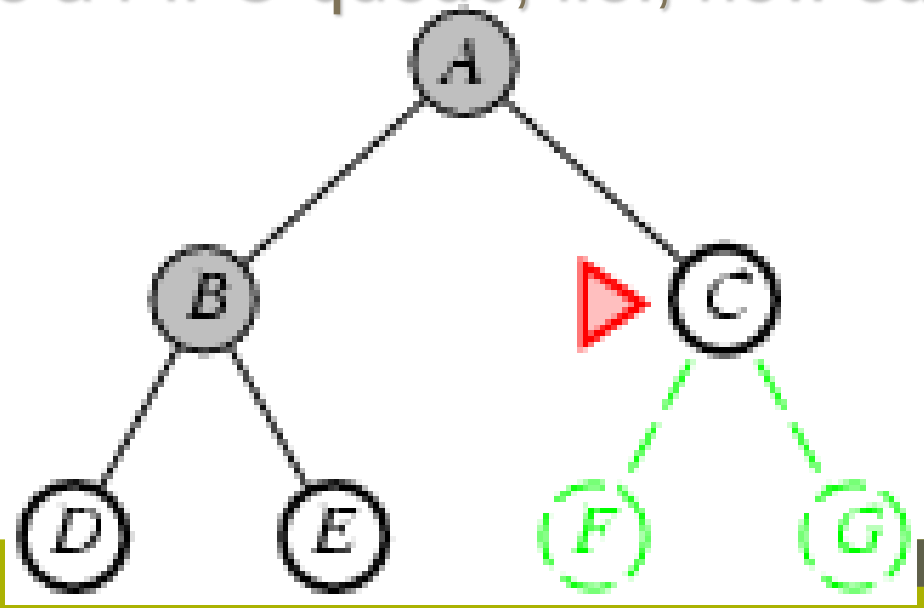


Breadth-first search

Expand shallowest unexpanded node

Implementation:

– *fringe* is a FIFO queue, i.e., new successors go at end

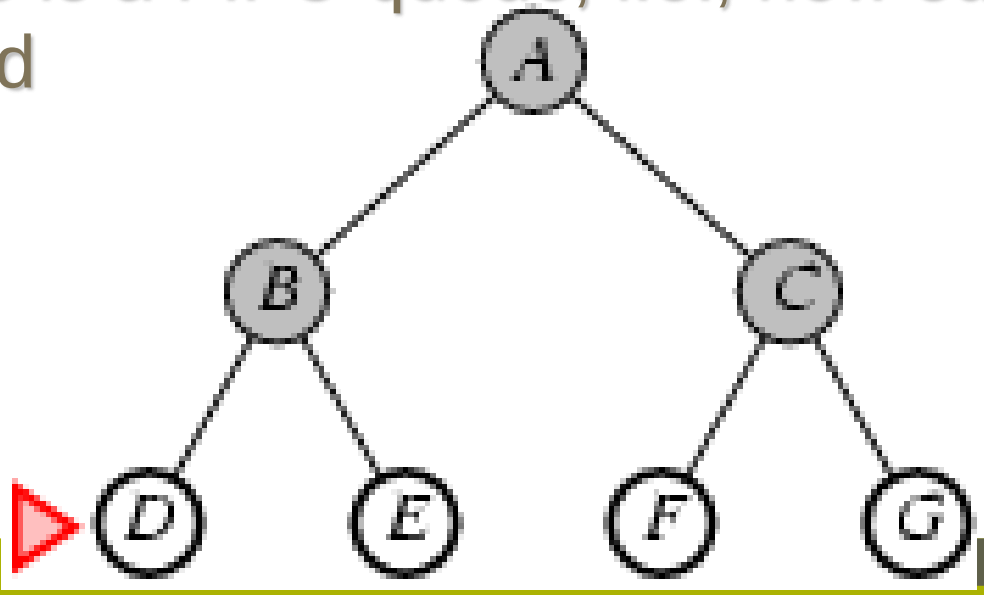


Breadth-first search

Expand shallowest unexpanded node

Implementation:

– *fringe* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete? Yes (if b is finite)

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

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)

BFS; evaluation

Two lessons:

- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH2	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Table is based on: b = 10; 10000/second; 1000 bytes/node

Uniform-cost search

Extension of BFS:

- Expand node with *lowest path cost*

Implementation: *fringe* = queue ordered by path cost.

UCS is the same as BFS when all step-costs are equal.

Uniform-cost search

Completeness:

- YES, if step-cost > ε (small positive constant)

Time complexity:

- Assume C* the cost of the optimal solution.
- Assume that every action costs at least ε
- Worst-case: $O(b^{C^*/\varepsilon})$

Space complexity:

- The same as time complexity

Optimality:

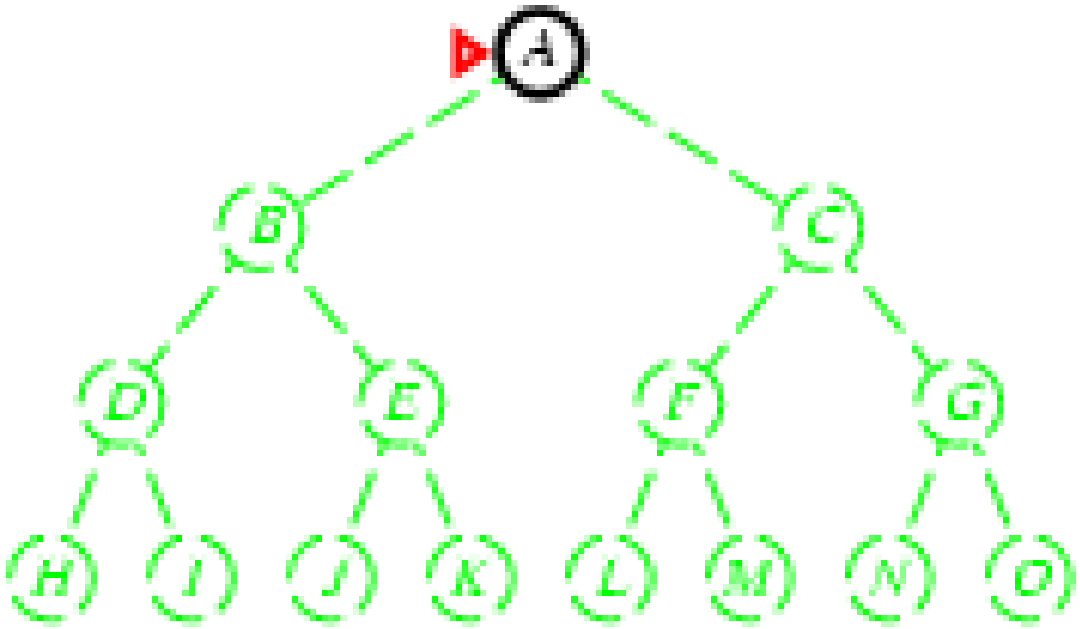
- nodes expanded in order of increasing path cost.
- YES, if complete.

Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

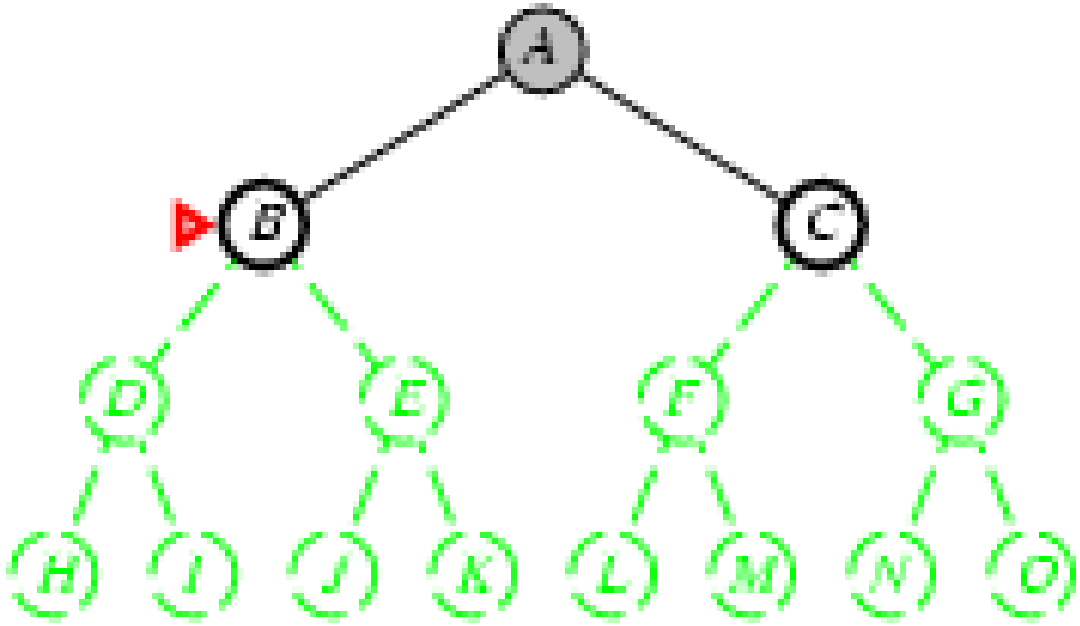


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

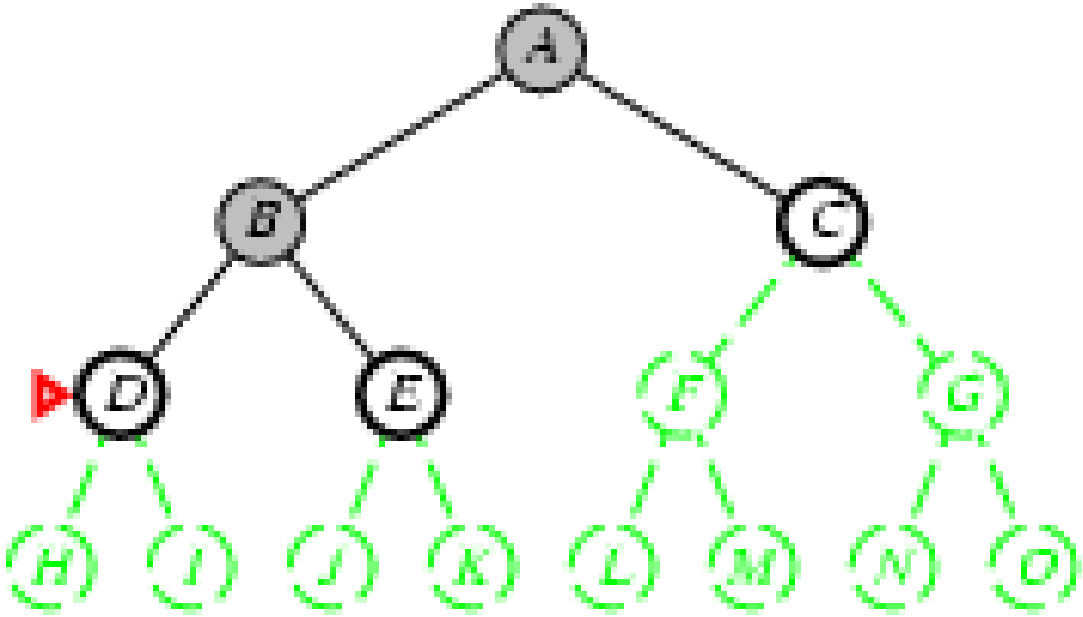


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

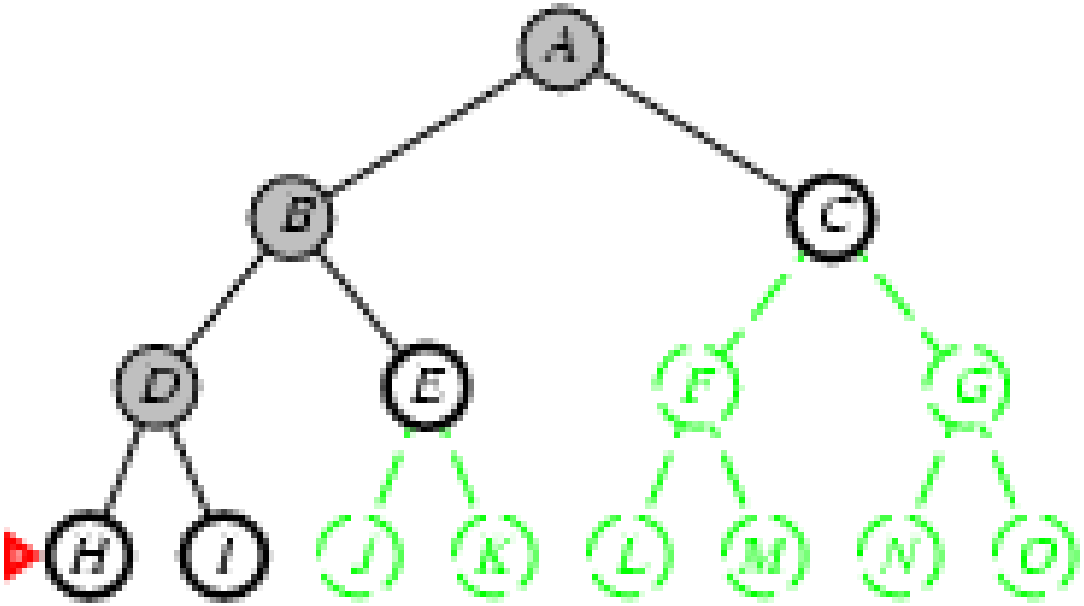


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

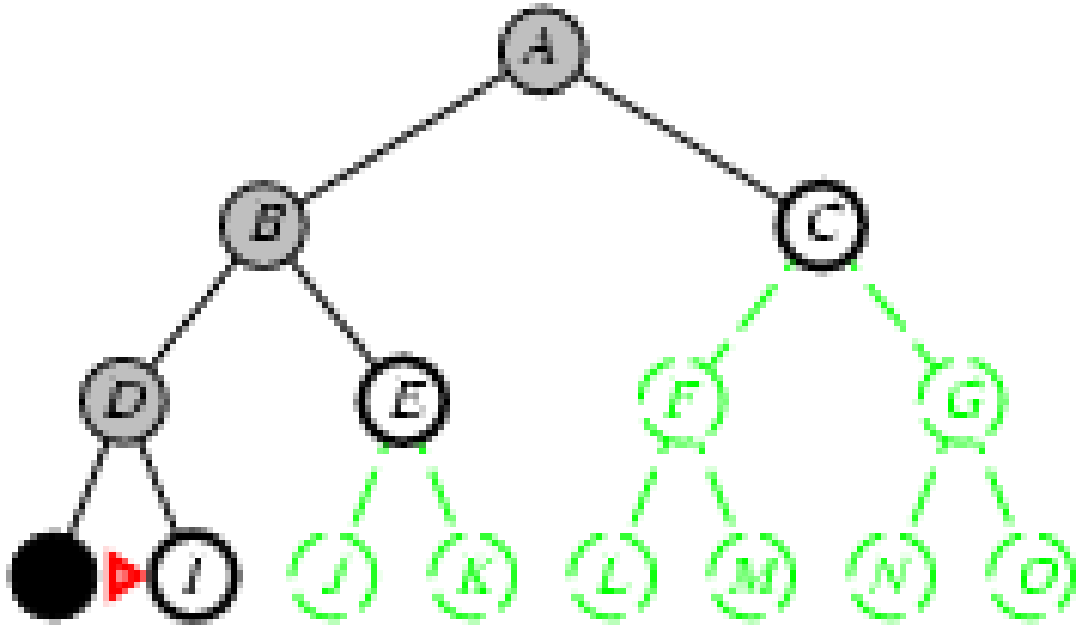


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

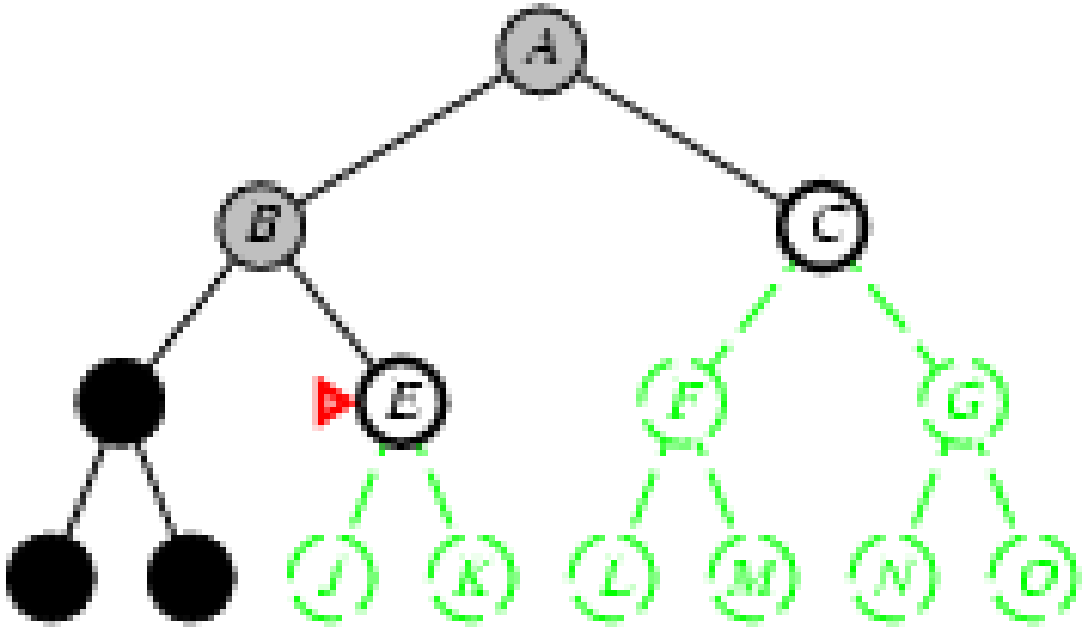


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

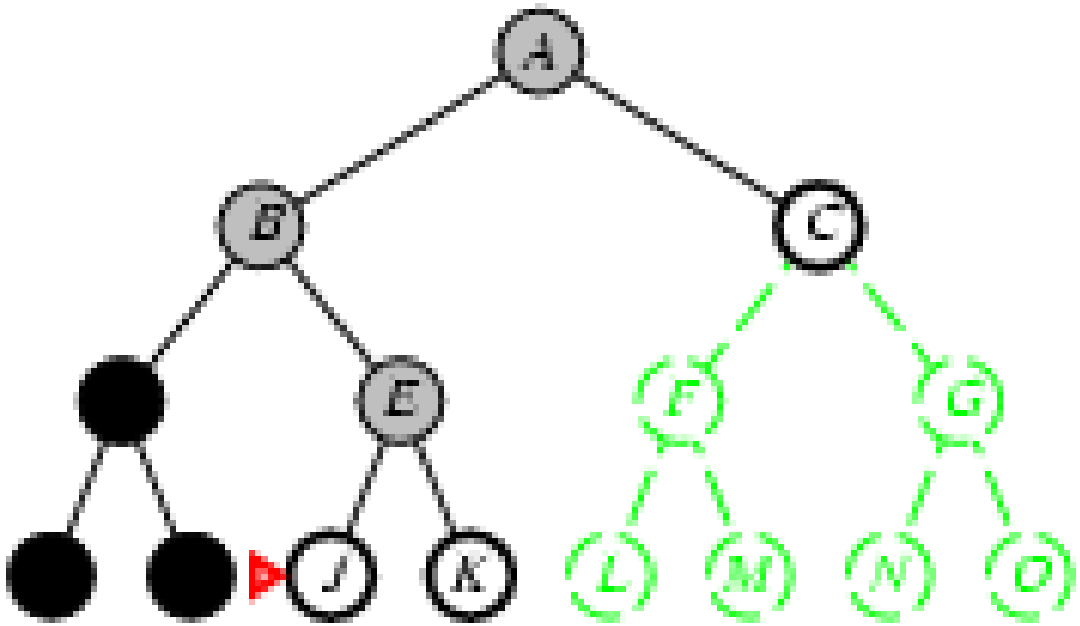


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

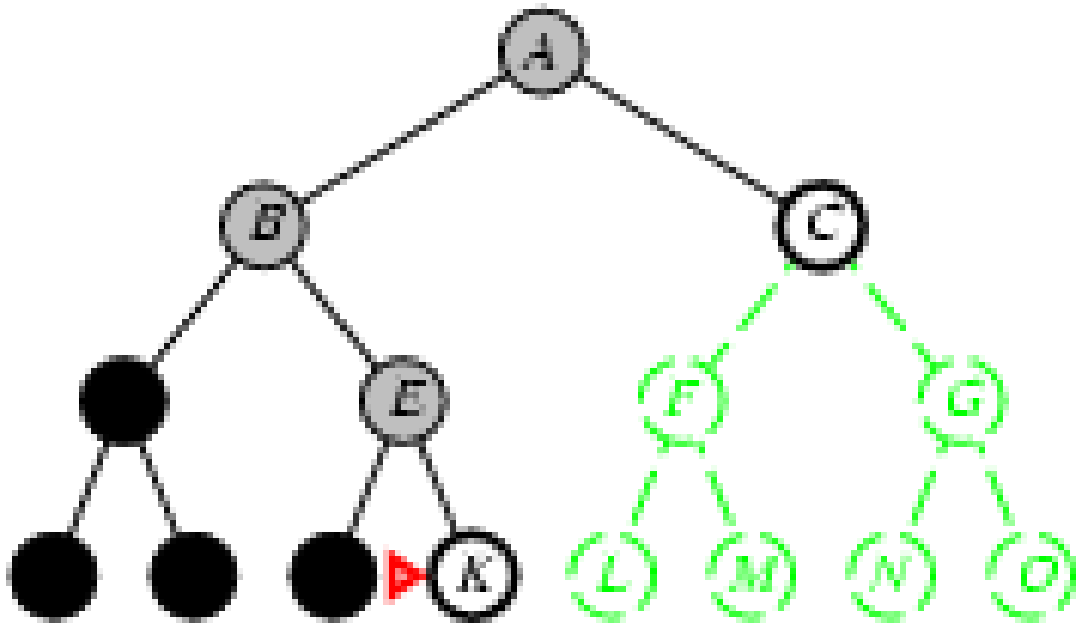


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

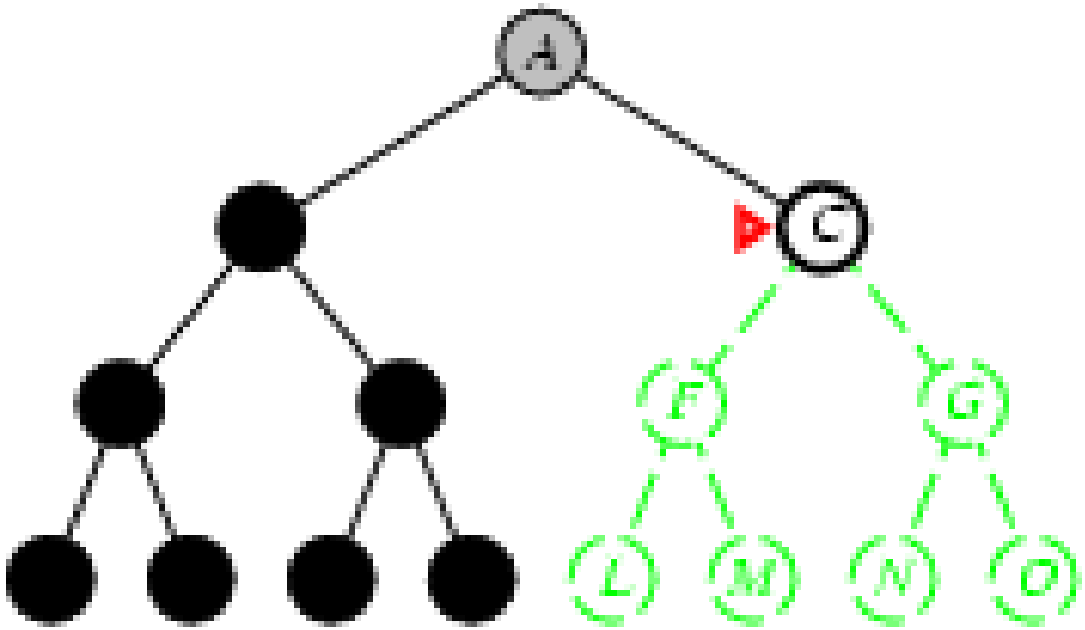


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

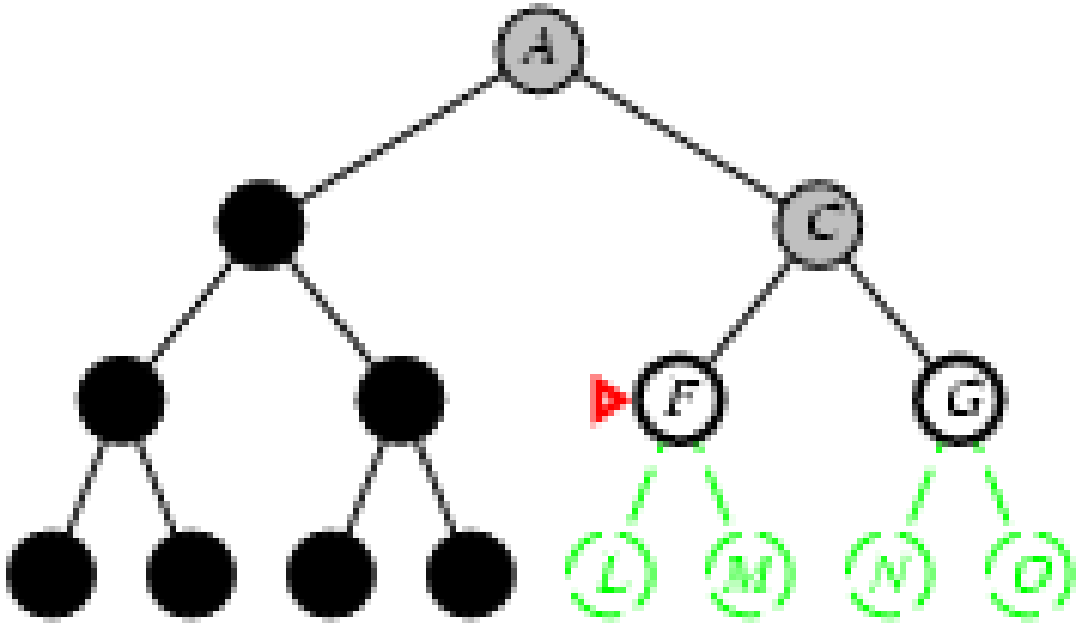


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

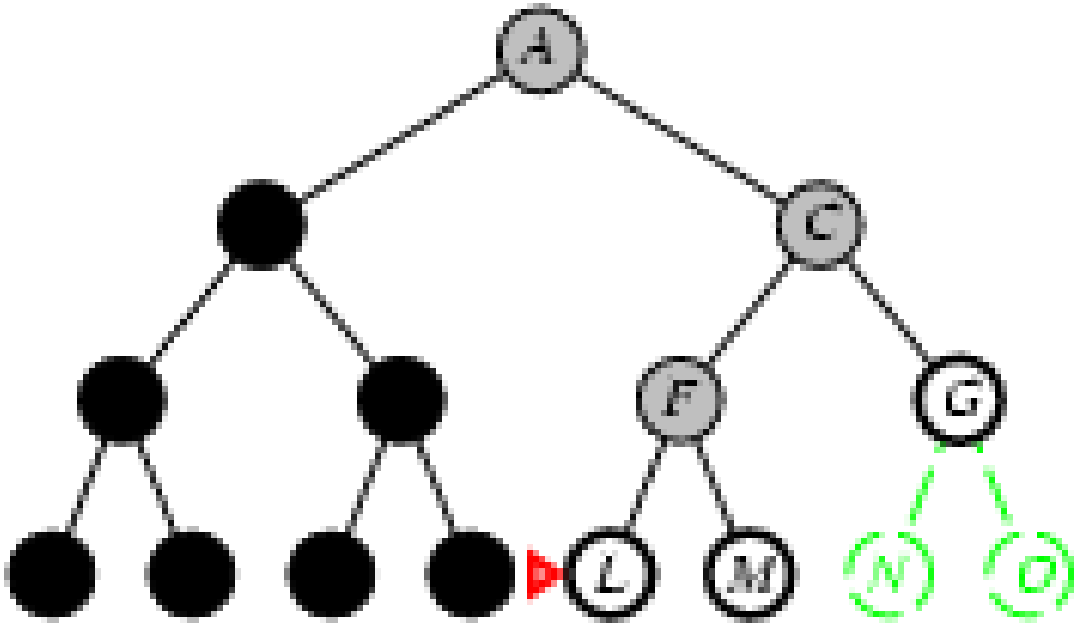


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front

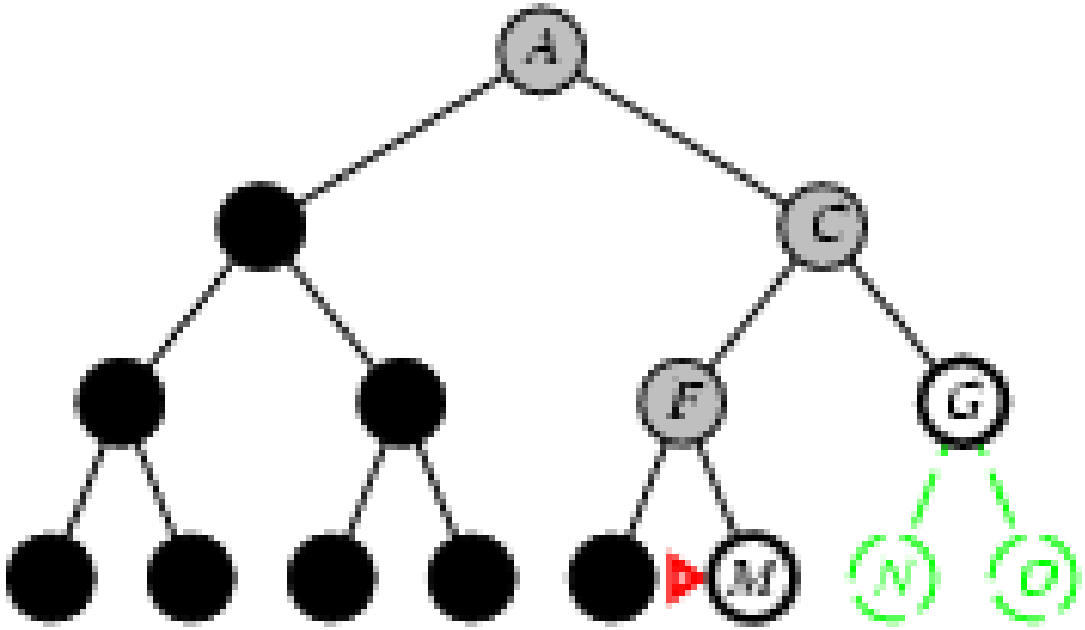


Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front



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

اگر با stack پیاده سازی کنیم bm و اگر recursive باشد
اندازه m می باشد (برای پیچیدگی حافظه)

اگر ابعاد درخت محدود باشد این الگوریتم complete می باشد ولی نمی توان حتی در این صورت هم تضمین کرد که این الگوریتم optimal می باشد .

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 search

= depth-first search with depth limit l,

i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Depth-limited search

Is DFS with depth limit *l*.

- i.e. nodes at depth *l* have no successors.
- Problem knowledge can be used

Solves the infinite-path problem.

If $l < d$ then incompleteness results.

If $l > d$ then not optimal.

Time complexity: $O(b^l)$

Space complexity: $O(bl)$

Iterative deepening search

```
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
```

Iterative deepening search

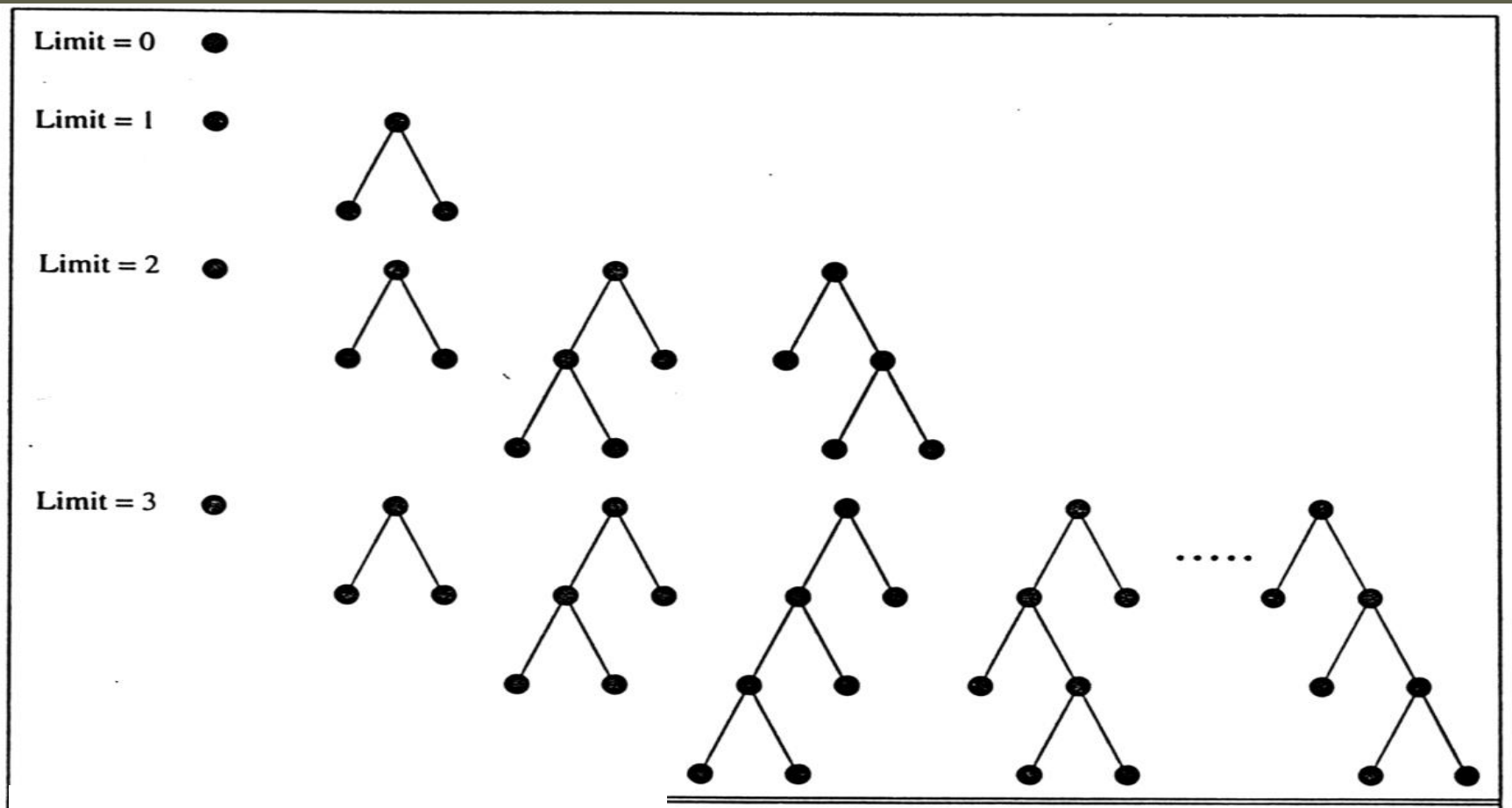


Figure 3.16 Four iterations of iterative deepening search on a binary tree.

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

ID search, evaluation

Completeness:

- YES (no infinite paths)

Time complexity: $O(b^d)$

- Algorithm seems costly due to repeated generation of certain states.
- Node generation:
  - level d: once
  - level d-1: 2
  - level d-2: 3
  - …
  - level 2: d-1
  - level 1: d

$$N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

Num. Comparison for b=10 and d=5 solution at far right

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Summary of algorithms

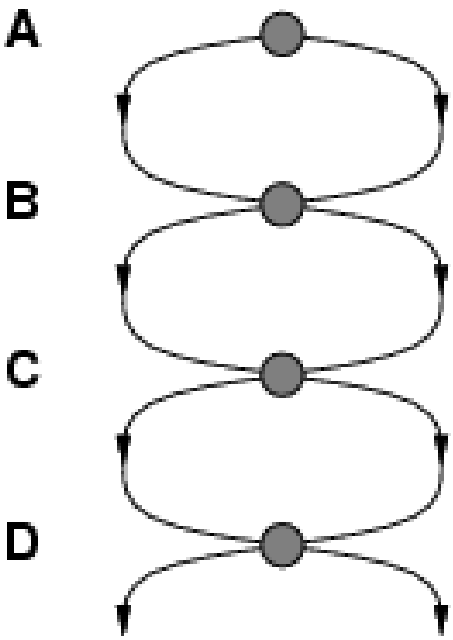
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

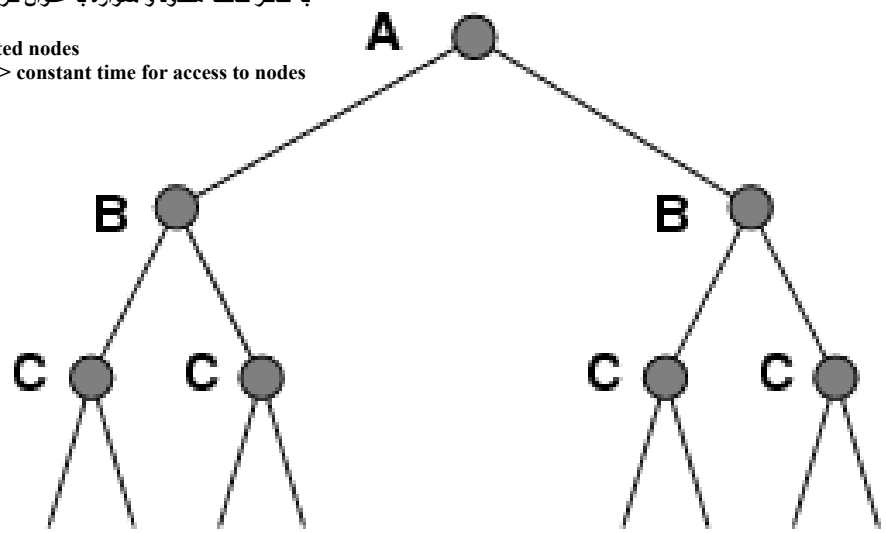
Repeated states

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

چرا در تمام مسائل گره های تکراری را نگه نمی داریم ؟
به خاطر حافظه محدود و همواره به عنوان طراح باید یک trade offبین حافظه و زمان برقرار سازد

open list : generated nodes
Close List : Expanded Nodes(HashTable) => constant time for access to nodes



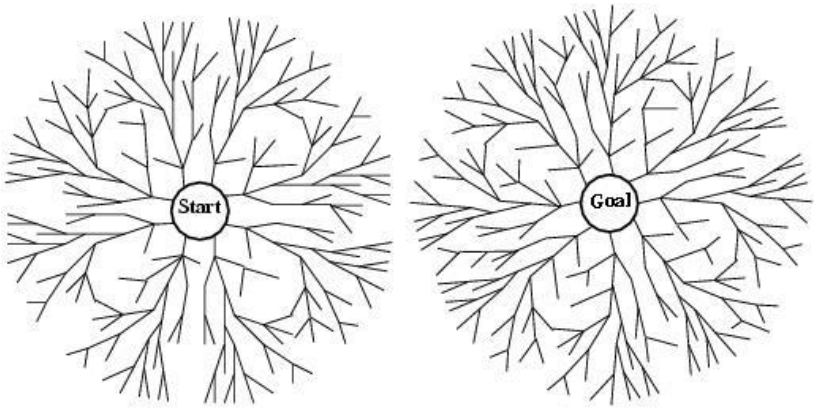


Graph search

```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Bidirectional search



Two simultaneous searches from start an goal.

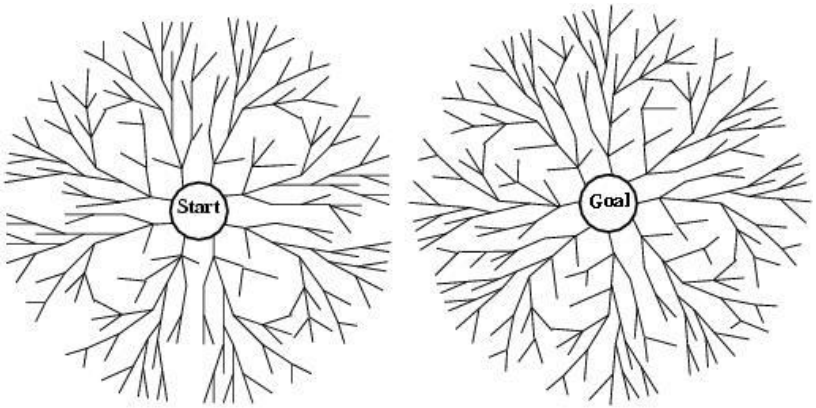
– Motivation: $b^{d/2} + b^{d/2} \neq b^{d}$

Check whether the node belongs to the other fringe before expansion.

Space complexity is the most significant weakness.

Complete and optimal if both searches are BFS.

How to search backwards?



The predecessor of each node should be efficiently computable.

- When actions are easily reversible.