Artificial Intelligence (AI)
“Uninformed Search Strategies”
Faculty of Computer and Information
Level three – Computer Science

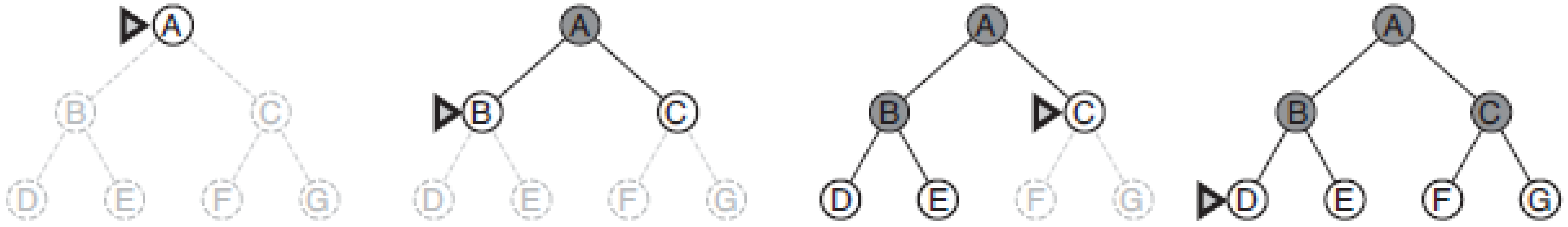
Dr. Mustafa Al-Sayed

(1) Breadth-first search (BFS)

- **Evaluation of BFS**; Optimal, Complete, and Exponential time and space
  - o BFS is complete algorithm as every node is expected to be explored if no solution in the shallowest levels
- **Strategy of BFS;** all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded (i.e., FIFO queue as a frontier).

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

BFS Example



Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- Suppose that root node generates ***b*** nodes at level#1, each of these nodes generate ***b*** nodes (for a total of b^2) at the level#2. Each of them generates ***b*** nodes (yielding b^3) at level#3, and so on.
- Suppose that solution is at depth ***d*** (i.e., level#d). In the worst case, the total number of nodes generated is $b + b^2 + b^3 + \cdot \cdot \cdot + b^d = O(b^d)$.
- Nodes at depth ***d*** would be expanded before the goal was detected. Therefore, the time complexity would be $O(b^{d+1})$.
- For space complexity, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier.

Both time and space complexity of BFS is scary, why?

- In sum, both time and space complexity is an exponential complexity bound such as ***$O(b^d)$*** is scary, where d is solution depth and b is the branching factor. The following table shows why?

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

- *The memory requirements are a bigger problem for breadth-first search than is the execution time*. One might wait 13 days for a solution with search depth 12, but no computer has the petabyte memory.
- If your problem has a solution at depth 16, then it will take 350 years for BFS. *In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

(2) Uniform-cost search

- **Evaluation of Uniform-cost**:
  - Optimal as it finds the smallest path cost
  - Complete when step cost exceeds a specific small cost
  - Exponential time and space complexity
- **Strategy of Uniform-Cost search**; when all step costs (cost of transition from node to node) are equal, BFS is optimal as it always expands the *shallowest* unexpanded nodes (i.e., nodes of the highest layers). Instead, uniform-cost search**:**
  1. Expands the node with the *lowest path cost*. This is done by storing the frontier as a priority queue ordered by the path cost (i.e., ordering of the queue by path cost)
  2. Goal test is applied to a node when it is *selected for expansion* rather than when it is first generated.
  3. A test is added in case a better path is found to a node currently on the frontier.

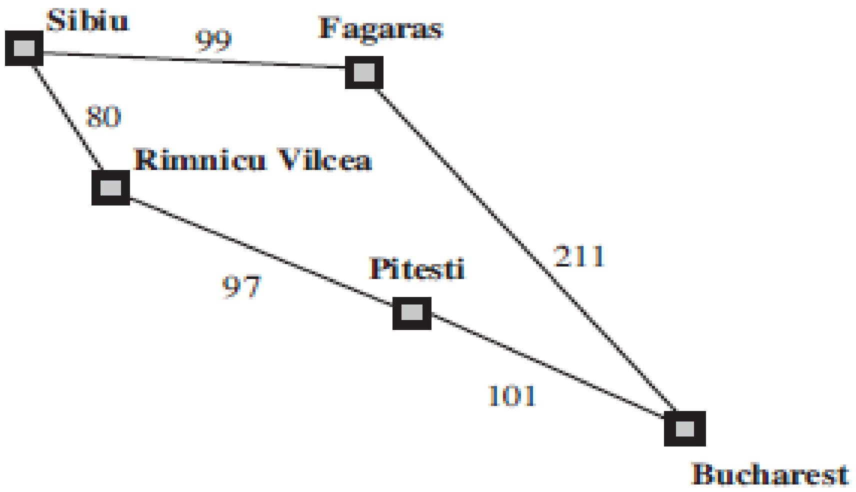
Uniform-cost search (Cont……)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

This algorithm is identical to the general graph search algorithm, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered

Uniform-cost search Example



- To get from **Sibiu** to **Bucharest**, successors of Sibiu are Rimnicu and Fagaras with costs 80 and 99, respectively. The least-cost node, Rimnicu, is expanded next, adding Pitesti with cost 80 + 97=177.
- Least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old is discarded. Bucharest, now with g-cost 278, is selected for expansion and solution is returned.

Complexity of Uniform-Cost search

- Uniform-cost search does not care about the *number* of steps, but only about their total cost (guided by path costs rather than depths)➔ This algorithm will get stuck in an infinite loop if there is zero-cost step ➔ So **completeness** is guaranteed when the cost of every step exceeds some small positive constant **ε**.
- Let ***C**** be the cost of the optimal solution, and every step cost at least **ε** ➔ the algorithm's worst-case time and space complexity is

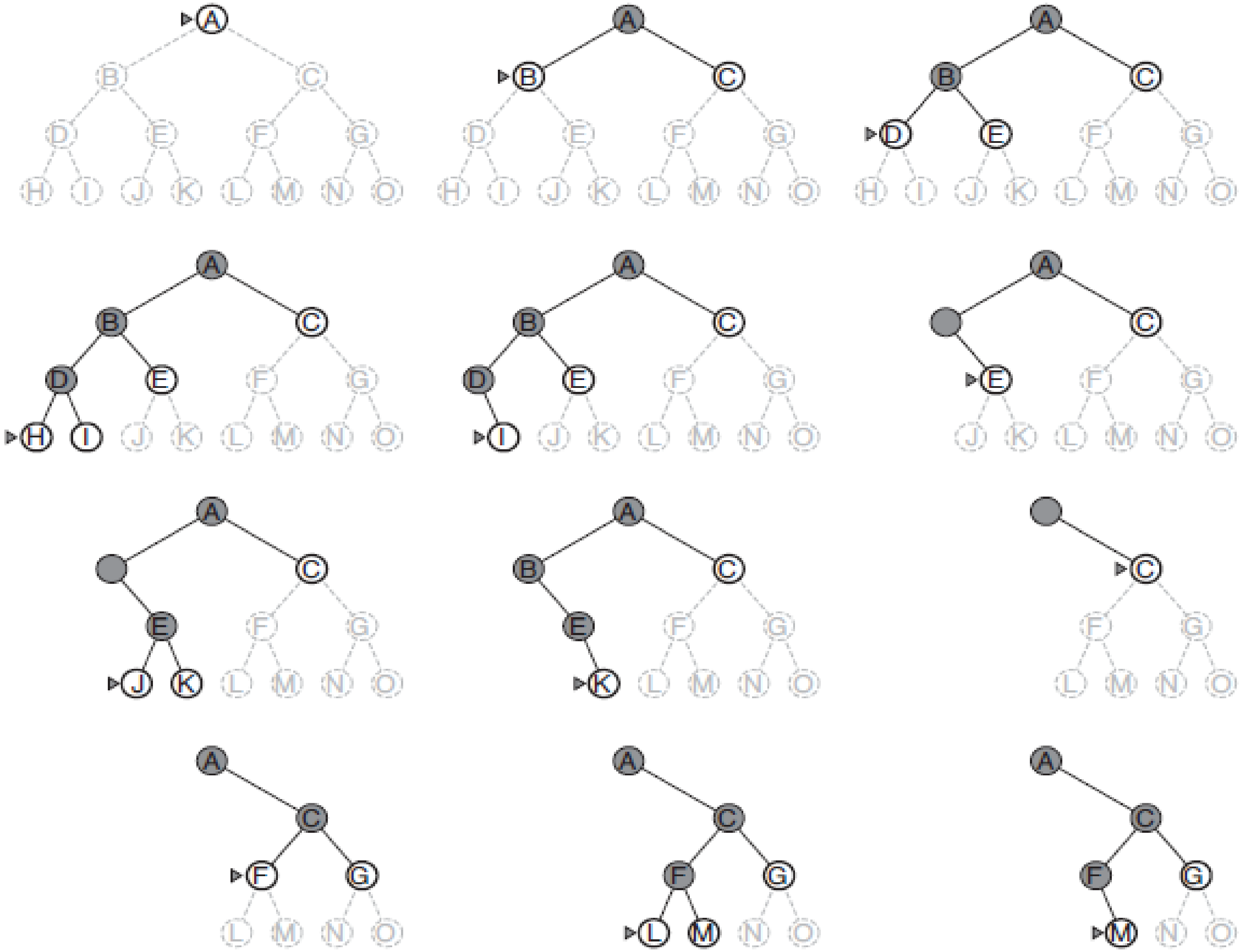
$$\boldsymbol{O}\left(\boldsymbol{b}^{\mathbf{1}+\left\lfloor \boldsymbol{C}^{*}/\in \right\rfloor}\right)$$

Where $\left\lfloor C^{*}/\in \right\rfloor$ = number nodes of the optimal path which can be much greater than ***d***. When all step costs are equal, $\left\lfloor C^{*}/\in \right\rfloor$ = d. When all step costs are the same, uniform-cost search is similar to breadth-first search

(3) Depth-first search (backtracking search) DFS

- **Evaluation of DFS**:
  - non-optimal as it will stop in case of detecting any solution that may not be the optimal one
  - complete as it will may expand every node but in case of avoiding repeated states as in the general graph algorithm instead of tree-algorithm
  - Exponential time complexity & linear space complexity
- **Strategy of DFS**:
  - It proceeds to the deepest level of the tree until nodes with no successors.
  - The expanded nodes are dropped from the frontier (LIFO queue).
  - If no solution, it “backs up” to the next deepest node that still has unexplored successors.
  - As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

DFS Example



DFS on a binary tree. Unexplored region is shown in light gray. Explored nodes with no descendants (children) in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Complexity of DFS

- As shown in the previous example, If node ***J*** were also a goal node, then DFS would return it as a solution instead of ***C***, which is the better solution➔ hence, depth-first search is not optimal.
- Graph-search version, which avoids repeated states and redundant paths, is complete.
- DFS in the worst case generates $\boldsymbol{O(b^m)}$ nodes, where m is the maximum depth in the tree.
- DFS has no clear advantage over BFS except in case of space complexity. DFS needs to store only a single path from the root to a leaf node that requires storage of only $\boldsymbol{O(bm)}$ nodes.

(4) Depth-limited search (DLS)

- **Strategy of DLS**:
  - Failure of DFS in infinite state spaces can be alleviated by supplying DFS with a predetermined depth limit *L* (i.e., nodes at depth *L* are treated as leaves).
  - DFS is a special case of depth-limited search with $\boldsymbol{L=\infty}$.
- **Evaluation of DFS**:
  - DLS is incompleteness if we choose $\boldsymbol{L < d}$ (d is depth of the shallowest goal).
  - DLS will be non-optimal if we choose $\boldsymbol{L > d}$.
  - Time complexity is $\boldsymbol{O(b^L)}$ and Space complexity is $\boldsymbol{O(bL)}$.

Depth-limited search (DLS) Cont….

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

- Note: DLS can terminate with two kinds of failure; the **standard failure value** indicates no solution, and ; **cutoff value** indicates no solution within the depth limit.

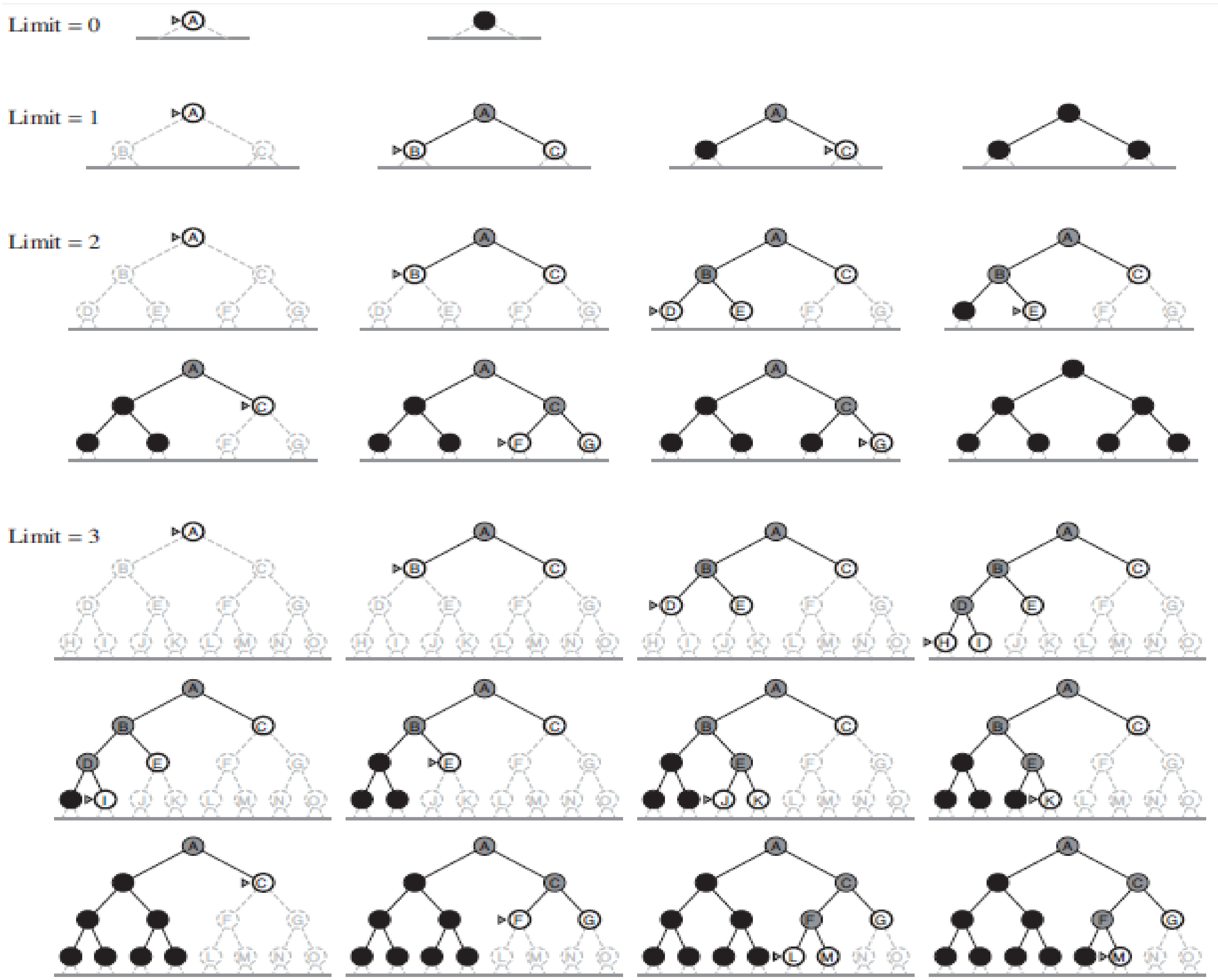
(5) Iterative deepening depth-first search (IDS)

- **Strategy of IDS**
  - IDS is used to finds the best depth limit.
  - Gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d (depth of the shallowest goal node).
  - IDS combines the benefits of DFS and BFS.
    - Like DFS, where space complexity ***O(bd)***
    - Like BFS, where the optimal solution is expected to be obtained

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

IDS algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns failure, meaning that no solution exists.

IDS Example



Does IDS wasteful due to regeneration of state multiple times?

- IDS not too costly:
  - Nodes on bottom level (depth d), which represent the majority, are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root that are generated d times (one for each iteration).
  - So total number of nodes generated in the worst case is

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d = O(b^d)$$

Compared to $N(\text{BFS}) = b + b^2 + \ldots + b^d = O(b^d)$

➔ Time complexity for IDS is the same as BFS with some extra cost. For example, if b = 10 and d = 5, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$$

$$N(\text{BFS}) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$$

- *In general, iterative deepening (IDS) is the preferred uninformed search method when the search space is large and the depth of the solution is not known.*

(6) Bidirectional search

- **Strategy:**
  - Run two simultaneous searches (i.e., one forward from the initial state and the other backward from the goal) hoping that the two searches meet in the middle.
  - The motivation is that $\boldsymbol{b^{d/2}} + \boldsymbol{b^{d/2}}$ is much less than $\boldsymbol{b^{d}}$. Therefore space and time complexity is $\boldsymbol{O(b^{d/2})}$.

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.