DEPTH-FIRST SEARCH

DEPTH-FIRST SEARCH

- Always expands the deepest node in the current frontier of the search tree.

- Depth-first search uses LIFO queue whereas Breadth-first search uses FIFO queue.

- Recursive function that calls itself on each of its children in turn.

DEPTH-FIRST SEARCH

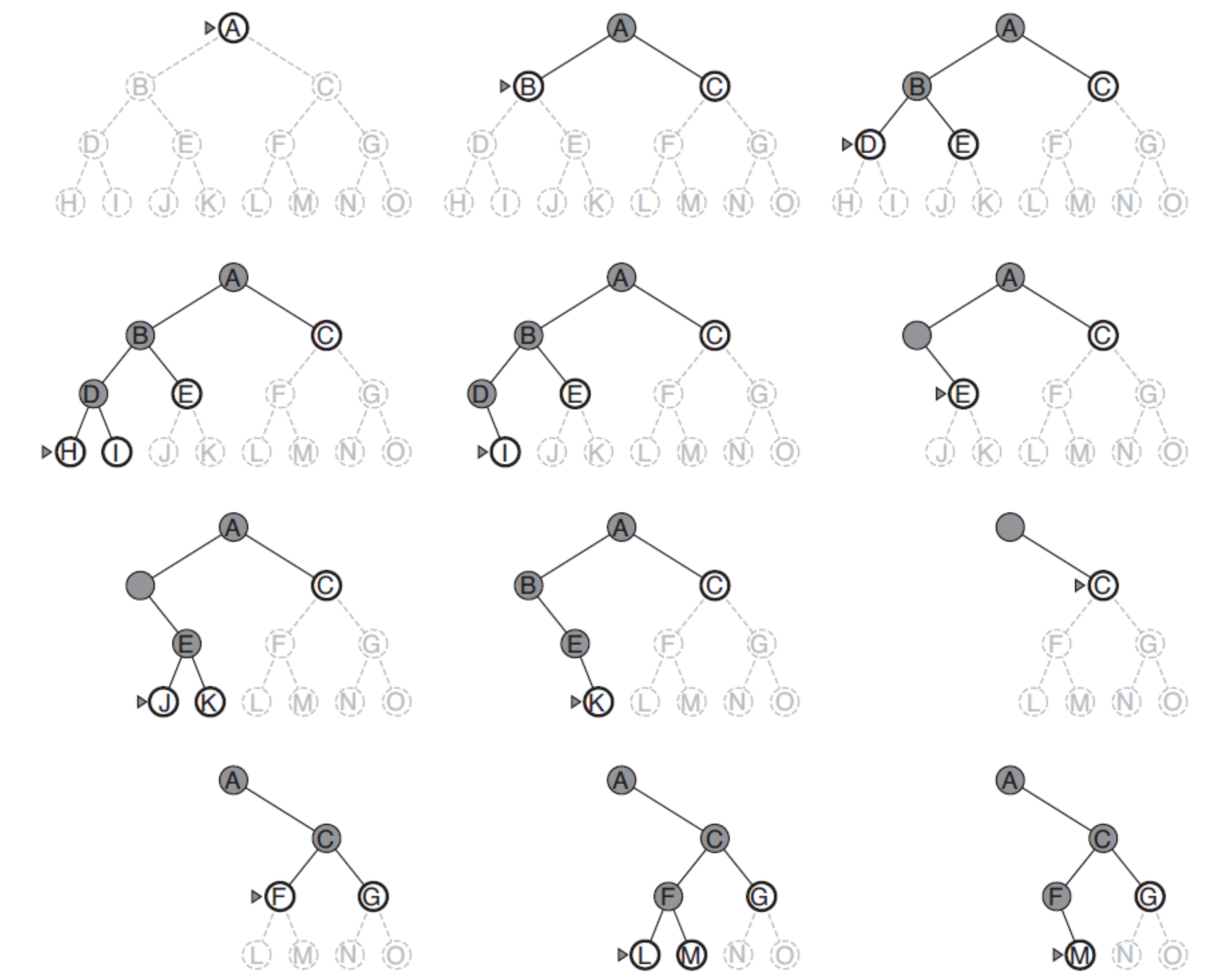
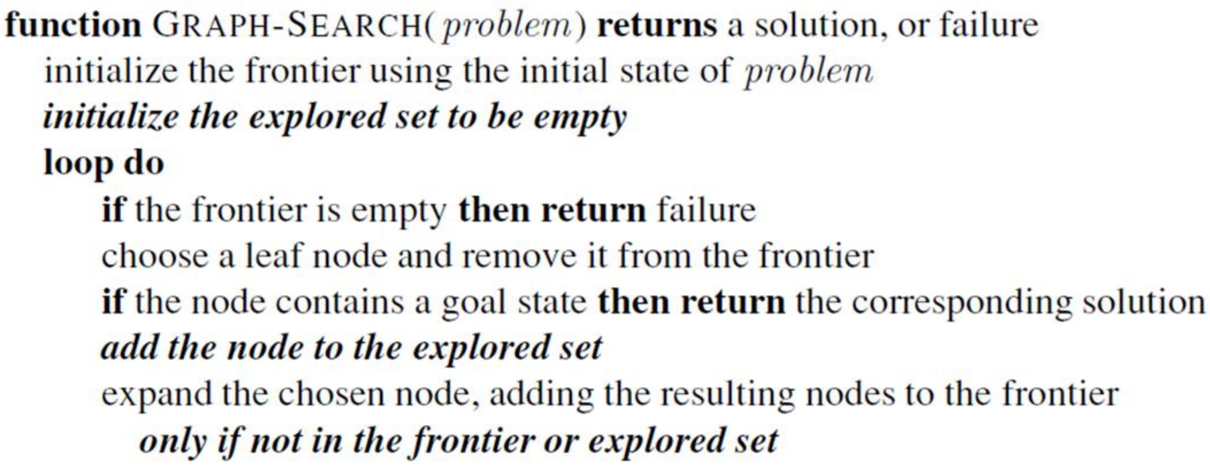

A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G I J K L M N O
A C E F G J K L M N O
A C E F G J K L M N O
A B C E F G K L M N O
C F G L M N O
A C F G L M N O
A C F G L M N O
A C F G M N O

Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

DEPTH-FIRST SEARCH

Graph-search version

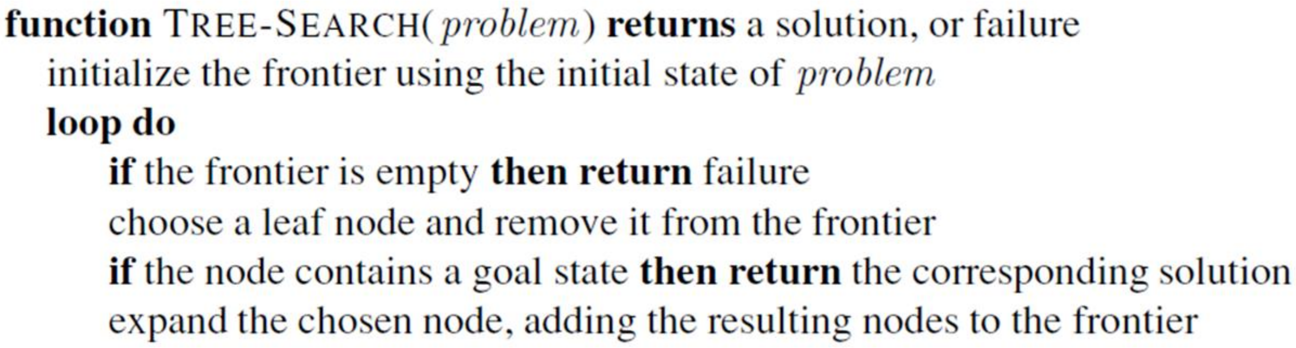
- Avoids repeated states and redundant path.
- Complete in finite space.



```
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
```

Tree-search version

- Not complete because of loops.
- Arad – Sibiu – Arad – Sibiu



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  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
     expand the chosen node, adding the resulting nodes to the frontier
```

CHALLENGES

- Non-optimal
- Searches complete path even if goal node may be very close to root node in another branch.

TIME COMPLEXITY

- O(b^m)
- m is maximum depth of any node

- For tree-search implementation, space complexity is O(bm)

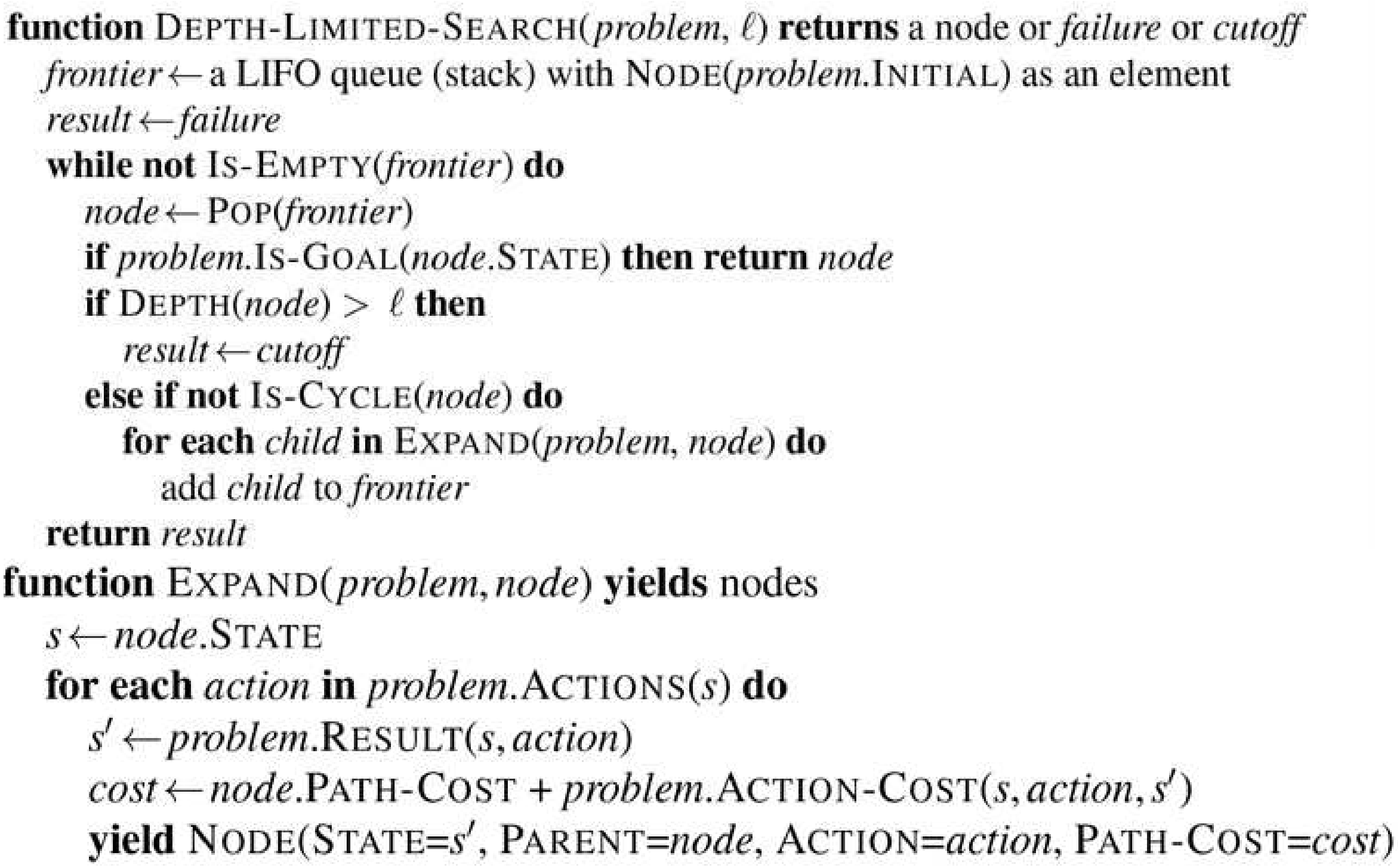
- Backtracking search variant has space complexity of O(m) due to expansion of only one successor.

DEPTH-LIMITED SEARCH

- To avoid failure of Depth-first search in infinite state space, we can predefine the depth limit l.

- Time complexity = $O(b^l)$
- Space complexity = $O(bl)$

DEPTH-LIMITED SEARCH



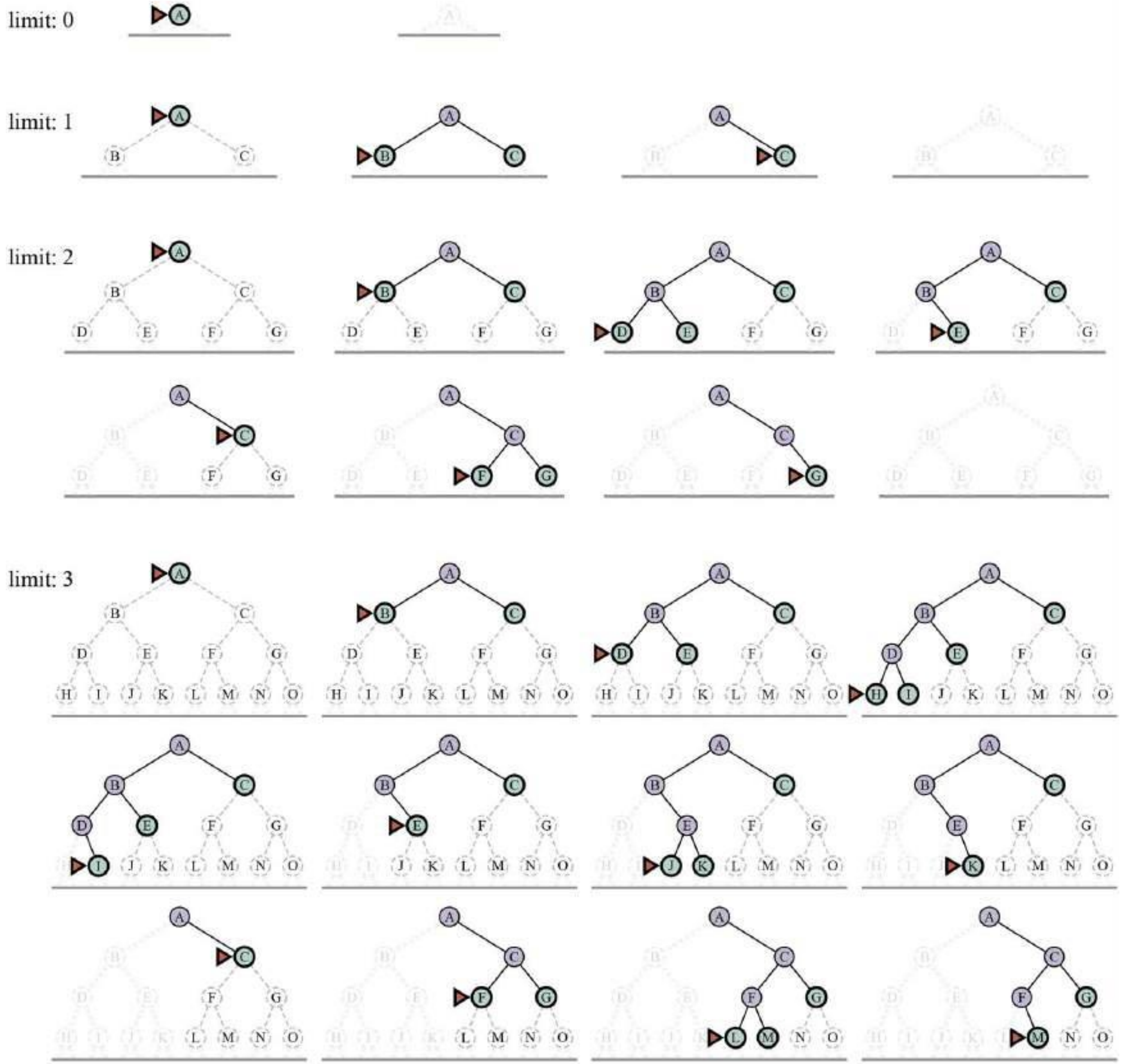
```
function Depth-Limited-Search(problem, $\ell$) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with Node(problem.Initial) as an element
  result ← failure
  while not Is-Empty(frontier) do
    node ← Pop(frontier)
    if problem.Is-Goal(node.State) then return node
    if Depth(node) > $\ell$ then
      result ← cutoff
    else if not Is-Cycle(node) do
      for each child in Expand(problem, node) do
        add child to frontier
  return result
function Expand(problem, node) yields nodes
  s ← node.State
  for each action in problem.Actions(s) do
    s′ ← problem.Result(s, action)
    cost ← node.Path-Cost + problem.Action-Cost(s, action, s′)
    yield Node(State=s′, Parent=node, Action=action, Path-Cost=cost)
```

DEPTH-LIMITED SEARCH

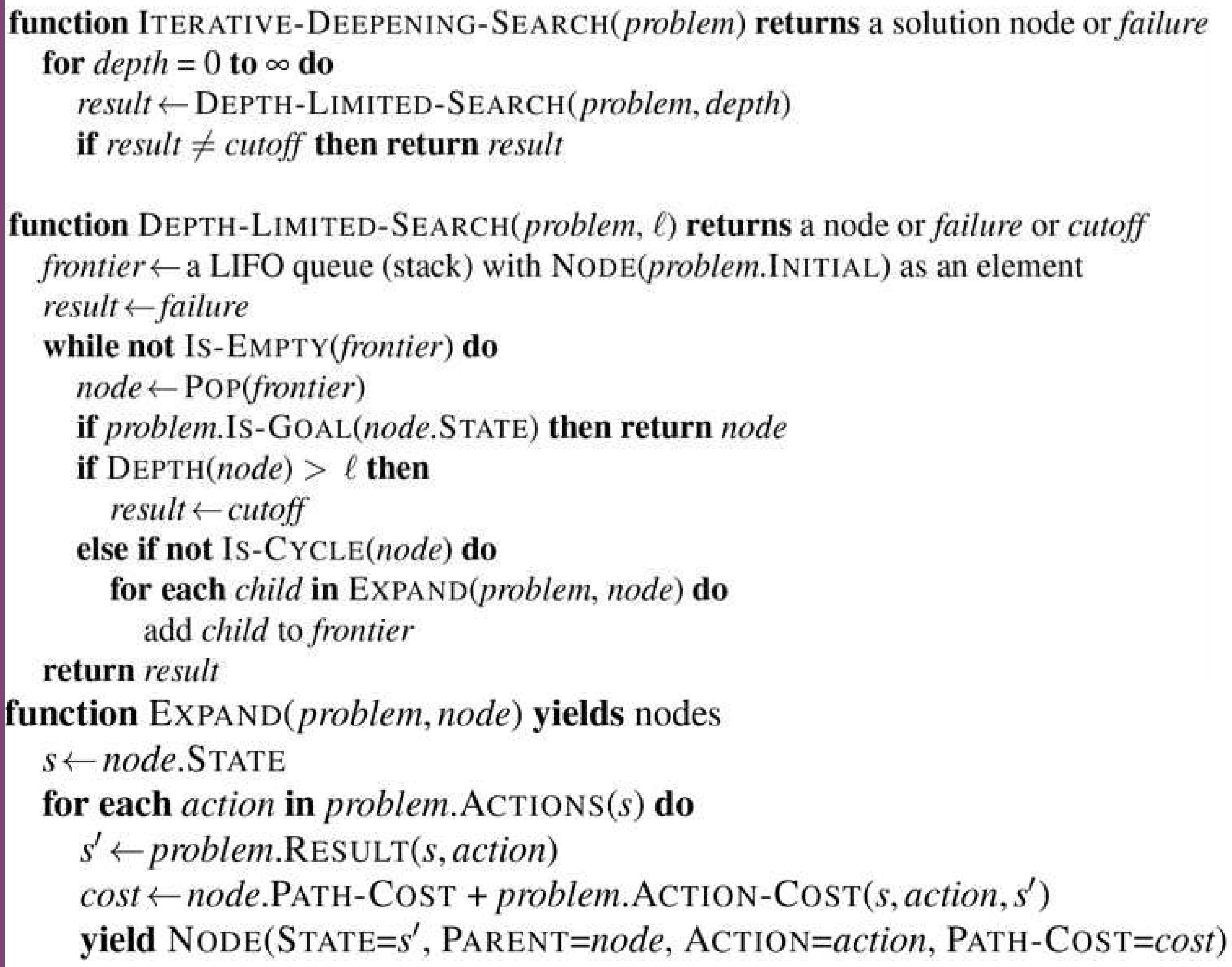
How to find the value of cut-off depth l ?

ITERATIVE DEEPENING SEARCH

GRADUALLY INCREASING THE DEPTH LIMIT UNTIL GOAL IS FOUND.



ITERATIVE DEEPENING SEARCH



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

function DEPTH-LIMITED-SEARCH(problem, $\ell$) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) $>$ $\ell$ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    $s'$ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, $s'$)
    yield NODE(STATE=$s'$, PARENT=node, ACTION=action, PATH-COST=cost)
```

ITERATIVE DEEPENING SEARCH

- Space complexity = O(bd)
- Time complexity = O(b^d) or O(b^m)

- Total number of nodes generated N (IDS) = $(d)b^1 + (d\text{-}1)b^2 + (d\text{-}2)b^3 + \ldots + b^d$

- Problem
  - Generates states multiple times.

HYBRID APPROACH

1. Run breath-first search until almost all the available memory is consumed.
2. Run iterative deepening search from all the nodes in the frontier.