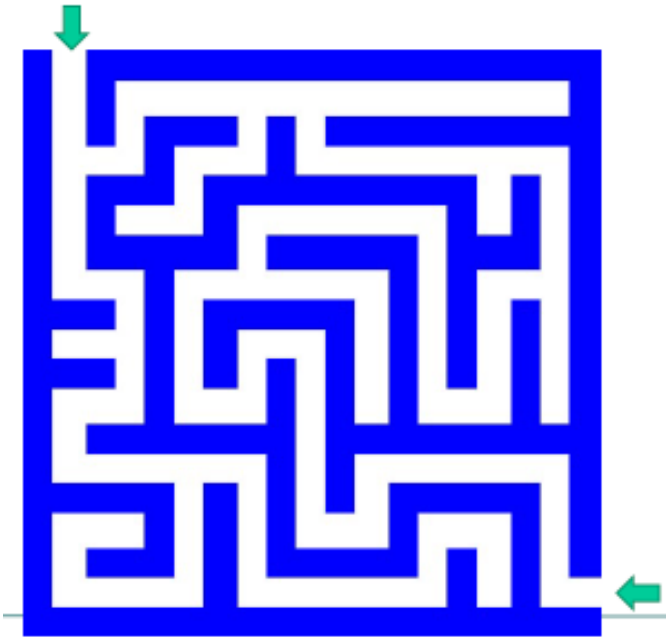


Artificial Intelligence

Lecture Eight: Blind Search

By: Dr. Zied O. Ahmed

LECTURE EIGHT: BLIND SEARCH

8.1 Depth-First and Breadth-First Search

The *Depth First Search* (DFS) is one of the most basic and fundamental Blind Search Algorithms.

It is for those who want to probe deeply down a potential solution path in the hope that solutions do not lie too deeply down the tree.

That is "DFS is a good idea when you are confident that all partial paths either reach dead ends or become complete paths after a reasonable number of steps.

In contrast, "DFS is a bad idea if there are long paths, even infinitely long paths, that neither reach dead ends nor become complete paths. To conduct a DFS:

(1) Put the Start Node on the list called OPEN.

(2) If OPEN is empty, exit with failure; otherwise continue.

(3) Remove the first node from OPEN and put it on a list called CLOSED.

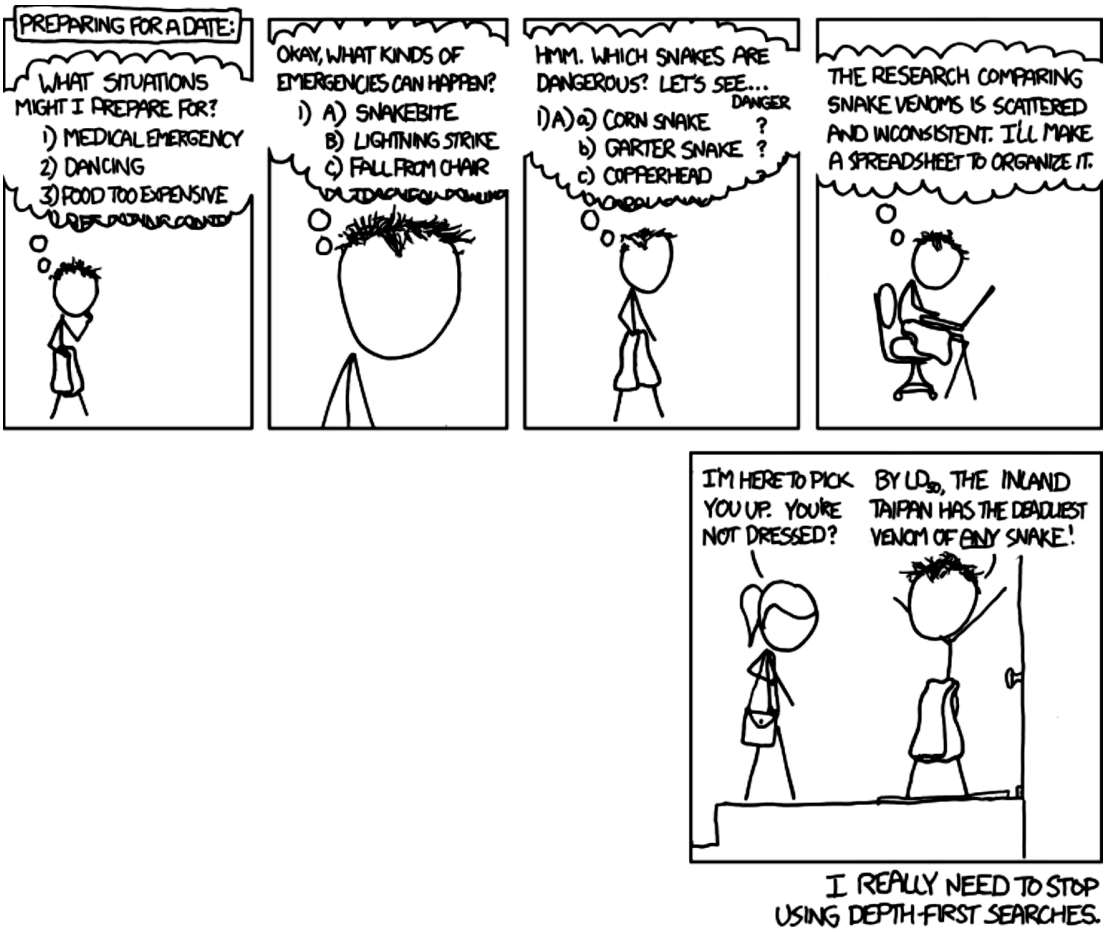
Call this node n.

(4) If the depth of n equals the depth bound, go to (2);

Otherwise continue.

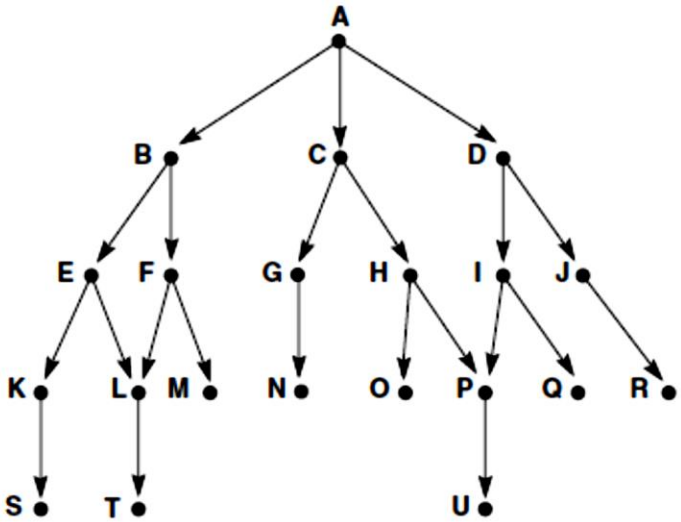
(5) Expand node n, generating all successors of n. Put these (in arbitrary order) at the beginning of OPEN and provide pointers back to n.

(6) If any of the successors are goal nodes, exit with the solution obtained by tracing back through the pointers; Otherwise go to (2).




```
function depth_first_search;
        begin
        open := [Start];                                        % initialize
        closed := [ ];_
        while open    [ ] do                                    % states remain
        begin
                remove leftmost state from open, call it X;
                if X is a goal then return SUCCESS              % goal found
                else begin
                   generate children of X;
                   put X on closed;
                   discard children of X if already on open or closed;     % loop check
                   put remaining children on left end of open % stack
                end
        end;
        return FAIL                                             % no states left
    end.
```

Consider the graph represented in figure below. States are labeled (A, B, C, . . .) so that they can be referred to in the discussion that follows.



In depth-first search, when a state is examined, all of its children and their descendants are examined before any of its siblings.

Depth-first search examines the states in the graph in the order A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R. The backtrack algorithm implemented depth-first search.

In this algorithm, the descendant states are added and removed from the *left* end of **open: open** is maintained as a *stack*, or last-in-first-out (LIFO) structure. The organization of **open** as a stack directs search toward the most recently generated states, Assume U is the goal state.

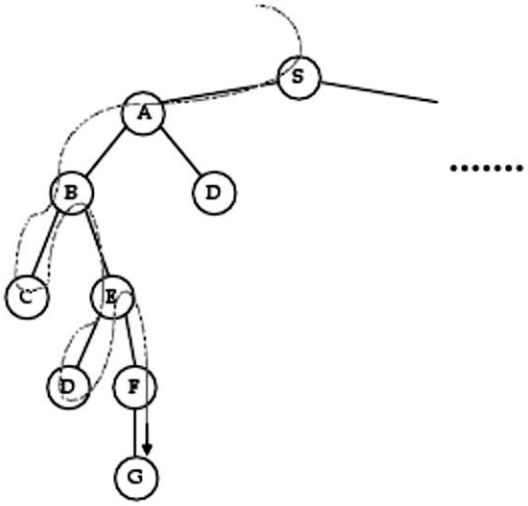
1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]

8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]

9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]

10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]

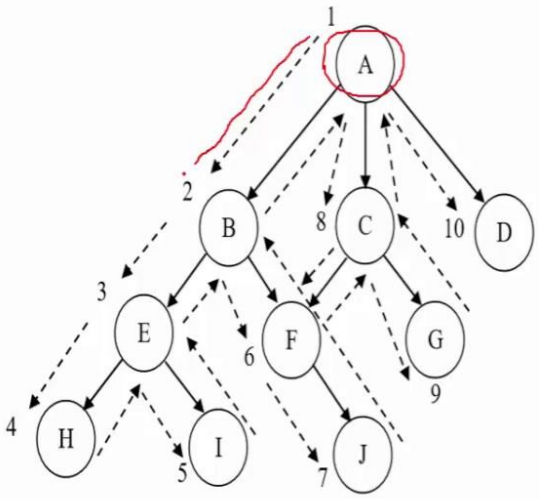
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

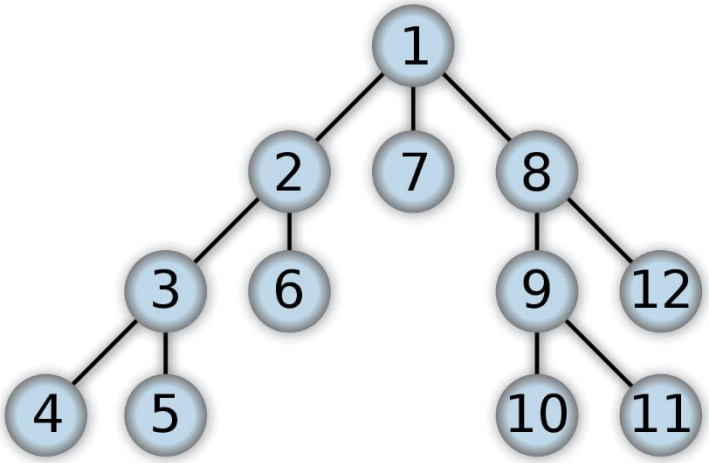


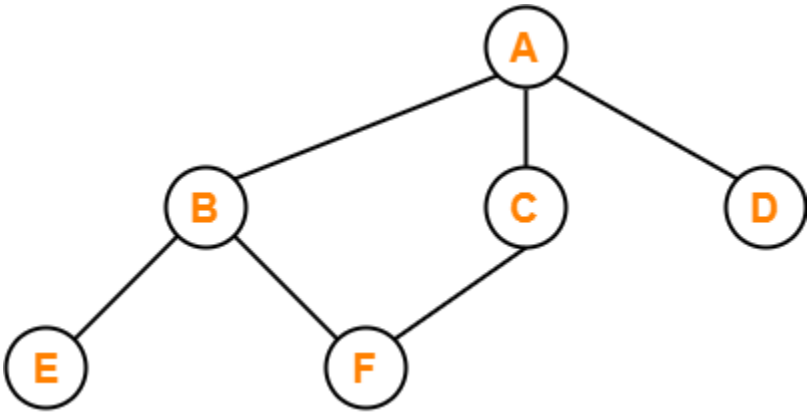
Examples:

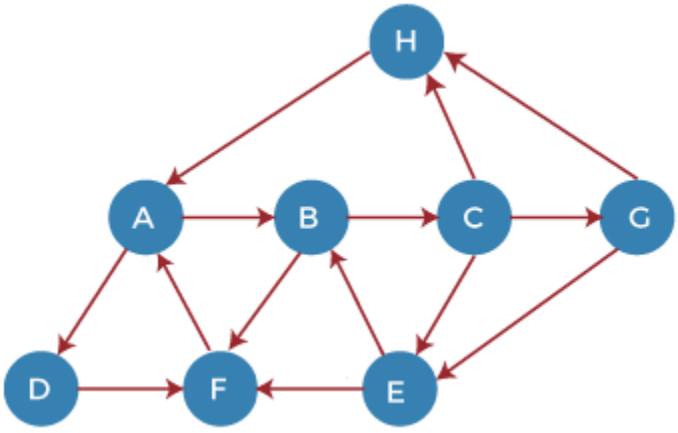
Q) Apply the depth first search algorithm on the following graph , where the start state is (A) and the desired goal state is (G),show the successive values of open and closed ,and the traversed path

Iteration #	X	open	closed
0	-	[A]	[ ]
1	A	[BCD]	[A]
2	B	[EFCD]	[BA]
3	E	[HIFCD]	[EBA]
4	H	[IFCD]	[HEBA]
5	I	[FCD]	[IHEBA]
6	F	[JCD]	[FIHEBA]
7	J	[CD]	[JFIHEBA]
8	C	[GD]	[CJFIHEBA]
9	G	G is the goal	




1
2
7
8
3
6
9
12
4
5
10
11


A
B
C
D
E
F


H
A
B
C
G
D
F
E

8.2 Breadth-First Search

Breadth First Search always explores nodes closest to the root node first, thereby visiting all nodes of a given length first before moving to any longer paths. It pushes uniformly into the search tree.

Breadth first search is most effective when all paths to a goal node are of uniform depth. It is a bad idea when the branching factor (average number of offspring for each node) is large or infinite.

Breadth First Search is also to be preferred over DFS if you are worried that there may be long paths (or even infinitely long paths) that neither reach dead ends or become complete paths.

The algorithm for Breadth First Tree Search is:

(1) Put the start node on a list called OPEN.

(2) If OPEN is empty, exit with failure;

Otherwise continue.

(3) Remove the first node on OPEN and put it on a list called CLOSED;

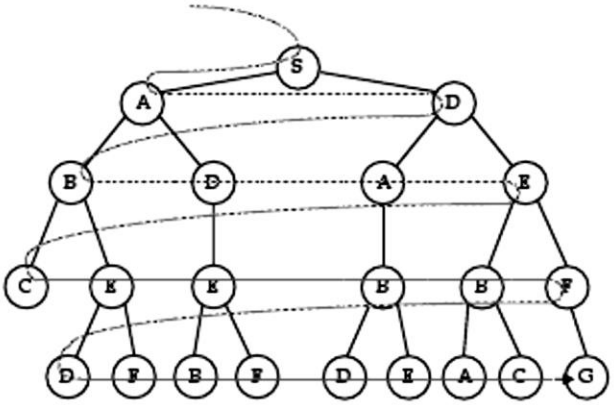
Call this node n;

(4) Expand node n, generating all of its successors. If there are no successors, go immediately to (2).

Put the successors at the end of OPEN and provdie pointers from these successors back to n.

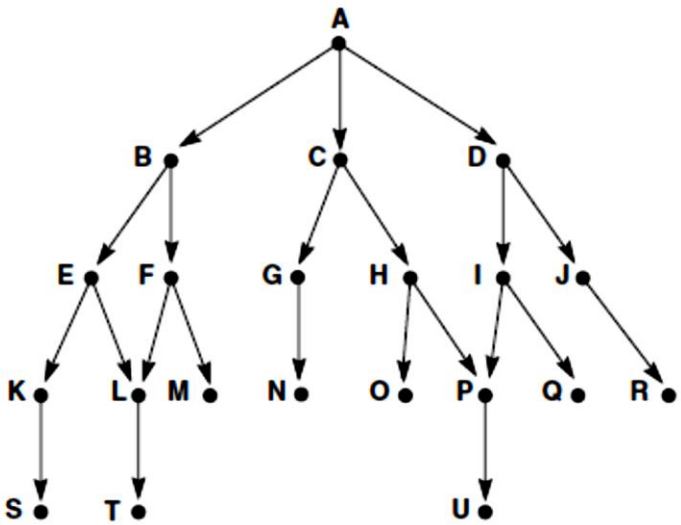
(5) If any of the successors are goal nodes, exit with the solution obtained by tracing back through the pointers;

Otherwise go to (2)




```
function breadth_first_search;
        begin
                open := [Start];                                        % initialize
                closed := [ ];_
                while open    [ ] do                                    % states remain
                  begin
                    remove leftmost state from open, call it X;
                    If X is a goal then return SUCCESS                  % goal found
                    else begin
                    generate children of X;
                    put X on closed;
                    discard children of X if already on open or closed;    % loop check
                    put remaining children on right end of open % queue
                   end
                end
                return FAIL                                             % no states left
        end.
```

Consider the graph represented in figure below. States are labeled (A, B, C, . . .) so that they can be referred to in the discussion that follows.



Breadth-first search, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next deeper level.

A breadth-first search of the graph considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

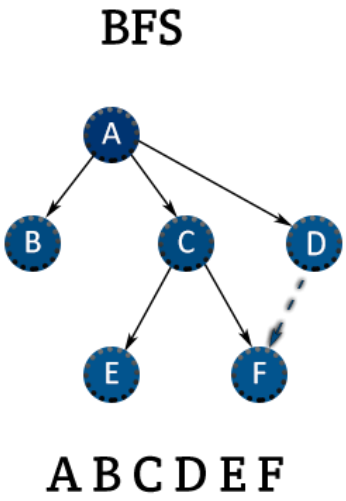
We implement breadth-first search using lists, **open** and **closed**, to keep track of progress through the state space.

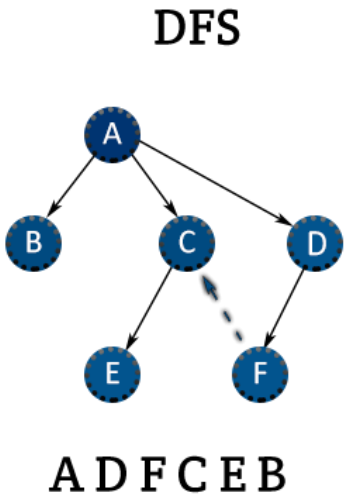
open, lists states that have been generated but whose children have not been examined. The order in which states are removed from **open** determines the order of the search. **closed** records states already examined.

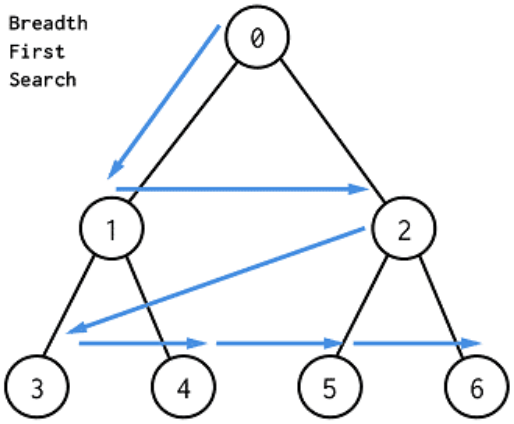
Each iteration produces all children of the state X and adds them to **open**. Note that **open** is maintained as a queue, or first-in-first-out (FIFO) data structure. States are added to the right of the list and removed from the left. These biases search toward the states that have been on open the longest, causing the search to be breadth-first.

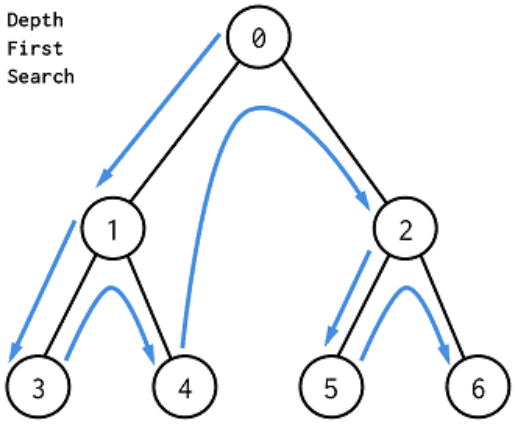
Child states that have already been discovered (already appear on either open or closed) are discarded. If the algorithm terminates because the condition of the "while" loop is no longer satisfied (**open** = [ ]) then it has searched the entire graph without finding the desired goal: the search has failed.

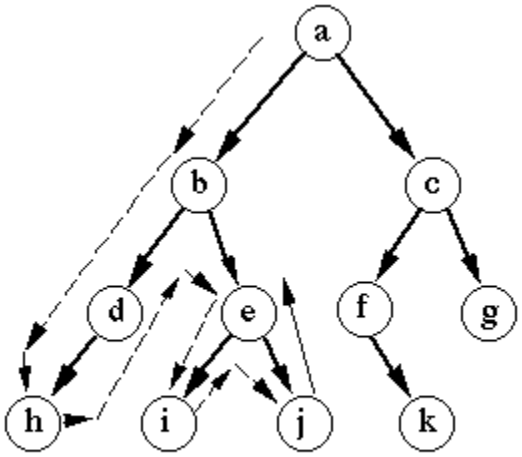
1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].



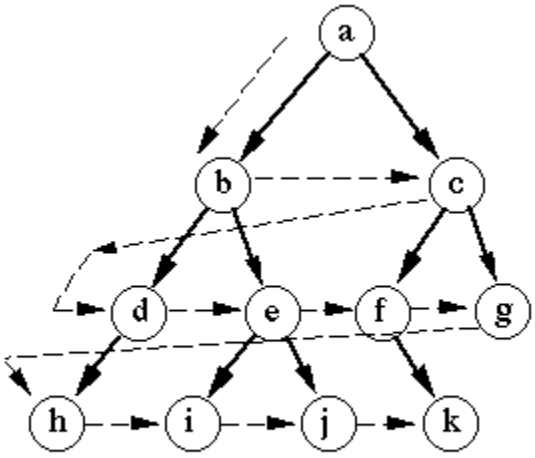








Depth-first search



Breadth-first search

BFS	DFS
BFS starts traversal from the root node and visits nodes in a level-by-level manner (i.e., visiting the ones closest to the root first).	DFS starts the traversal from the root node and visits nodes as far as possible from the root node (i.e., depth wise).
Usually implemented using a queue data structure.	Usually implemented using a stack data structure.
Generally, requires more memory than DFS.	Generally, requires less memory than BFS.

Optimal for finding the shortest distance.	Not optimal for finding the shortest distance.
Used for finding the shortest path between two nodes, testing if a graph is bipartite, finding all connected components in a graph, etc.	Used for topological sorting, solving problems that require graph backtracking, detecting cycles in a graph, finding paths between two nodes, etc.

8.3 Depth-limited search

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit $\mathscr{L}$. That is, nodes at depth $\mathscr{L}$ are treated as if they have no successors.

This approach is called **depth-limited search**. The depth limit solves the infinite-path problem.

Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm. Alternatively, it can be implemented as a simple recursive algorithm.

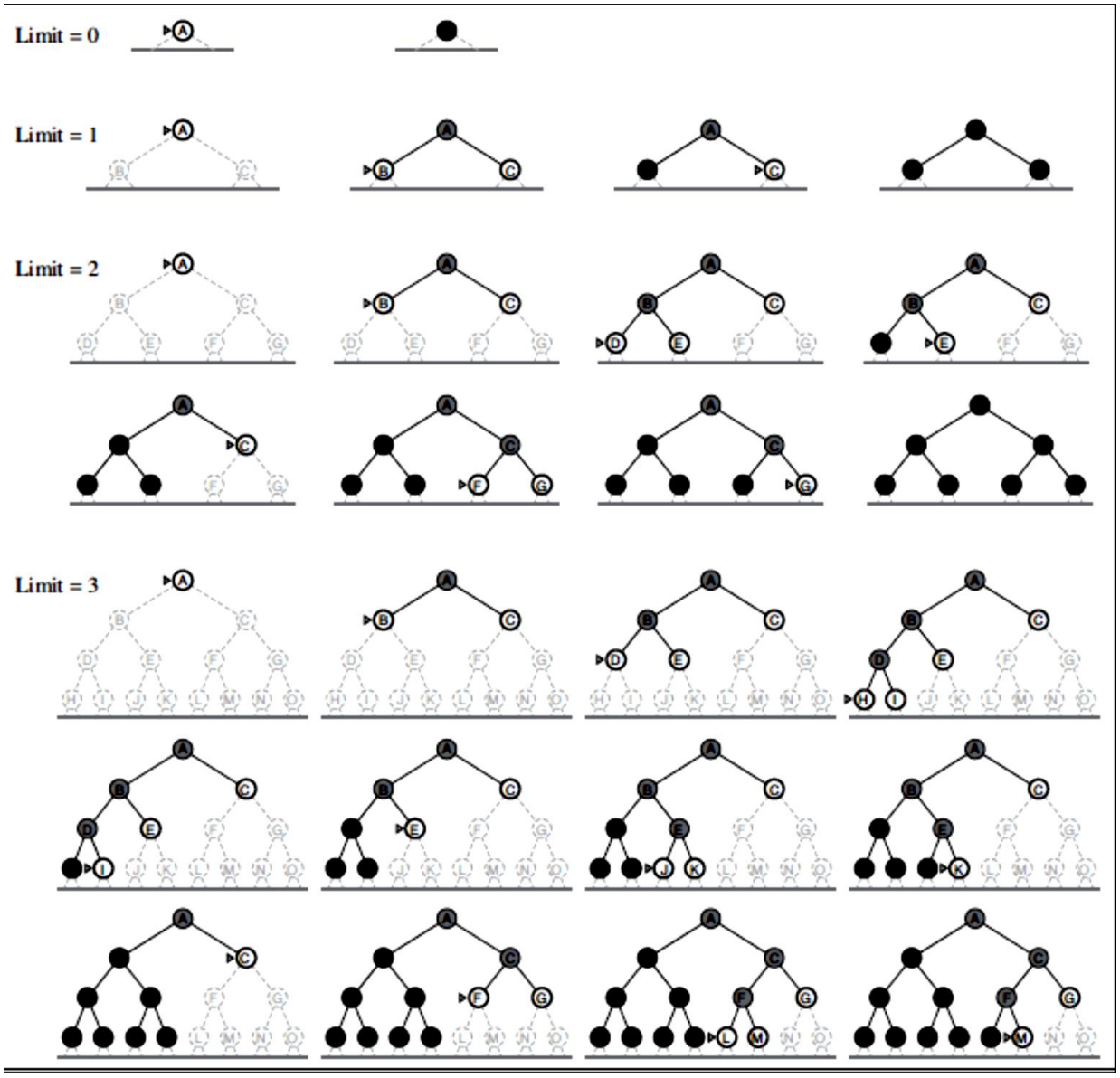
Notice that depth-limited search can terminate with two kinds of failure: the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.

8.4 Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with *depth-first tree search*, that finds the best depth limit. It does this by 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, the depth of the shallowest goal node.

Iterative deepening combines the benefits of *depth-first* and *breadth-first* search.


Limit = 0
Limit = 1
Limit = 2
Limit = 3