1.3.4.2 UNIFORM-COST SEARCH

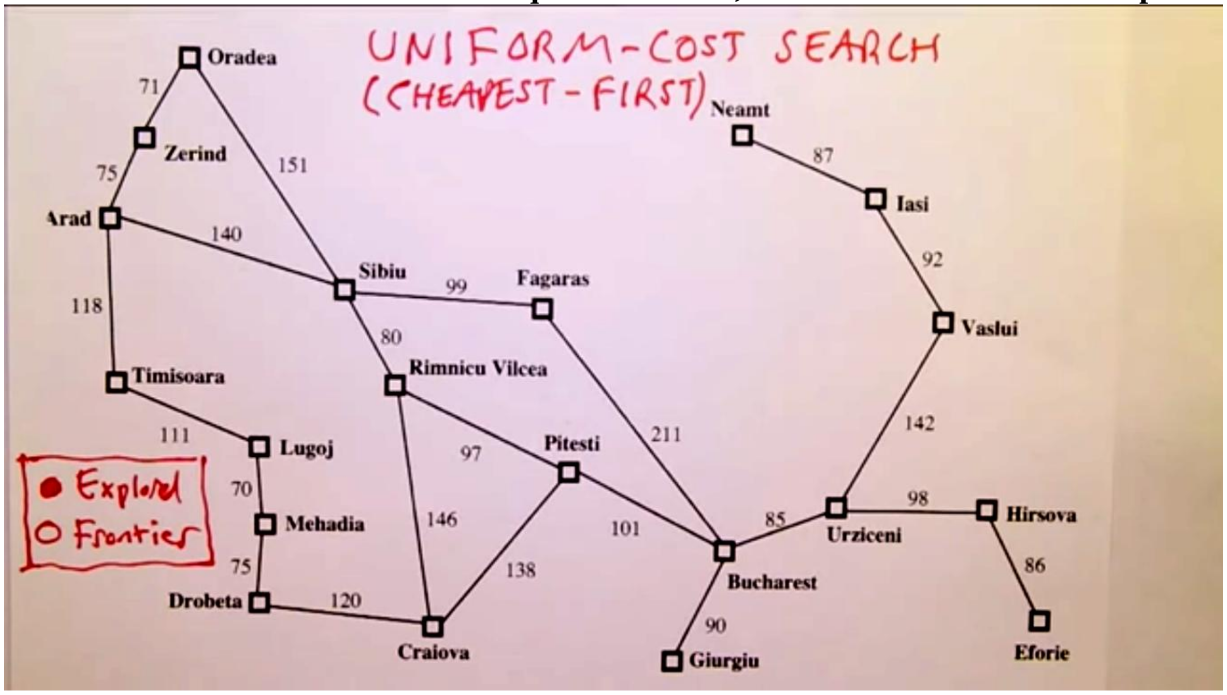
Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Uniform Cost Search

Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics. It can solve any general graph for optimal cost. Uniform Cost Search as it sounds searches in branches which are more or less the same in cost.

Uniform Cost Search again demands the use of a priority queue. Recall that Depth First Search used a priority queue with the depth upto a particular node being the priority and the path from the root to the node being the element stored. The priority queue used here is similar with the priority being the cumulative cost upto the node. Unlike Depth First Search where the maximum depth had the maximum priority, Uniform Cost Search gives the minimum cumulative cost the maximum priority. The algorithm using this priority queue is the following:

```
Insert the root into the queue
While the queue is not empty
    Dequeue the maximum priority element from the queue
    (If priorities are same, alphabetically smaller path is chosen)
    If the path is ending in the goal state, print the path and exit
    Else
        Insert all the children of the dequeued element, with the cumulative costs as priority
```



2.5.1.3 DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 1.31. The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors. As those nodes are expanded,they are dropped from the fringe,so then the search ―backs up‖ to the next shallowest node that still has unexplored successors.

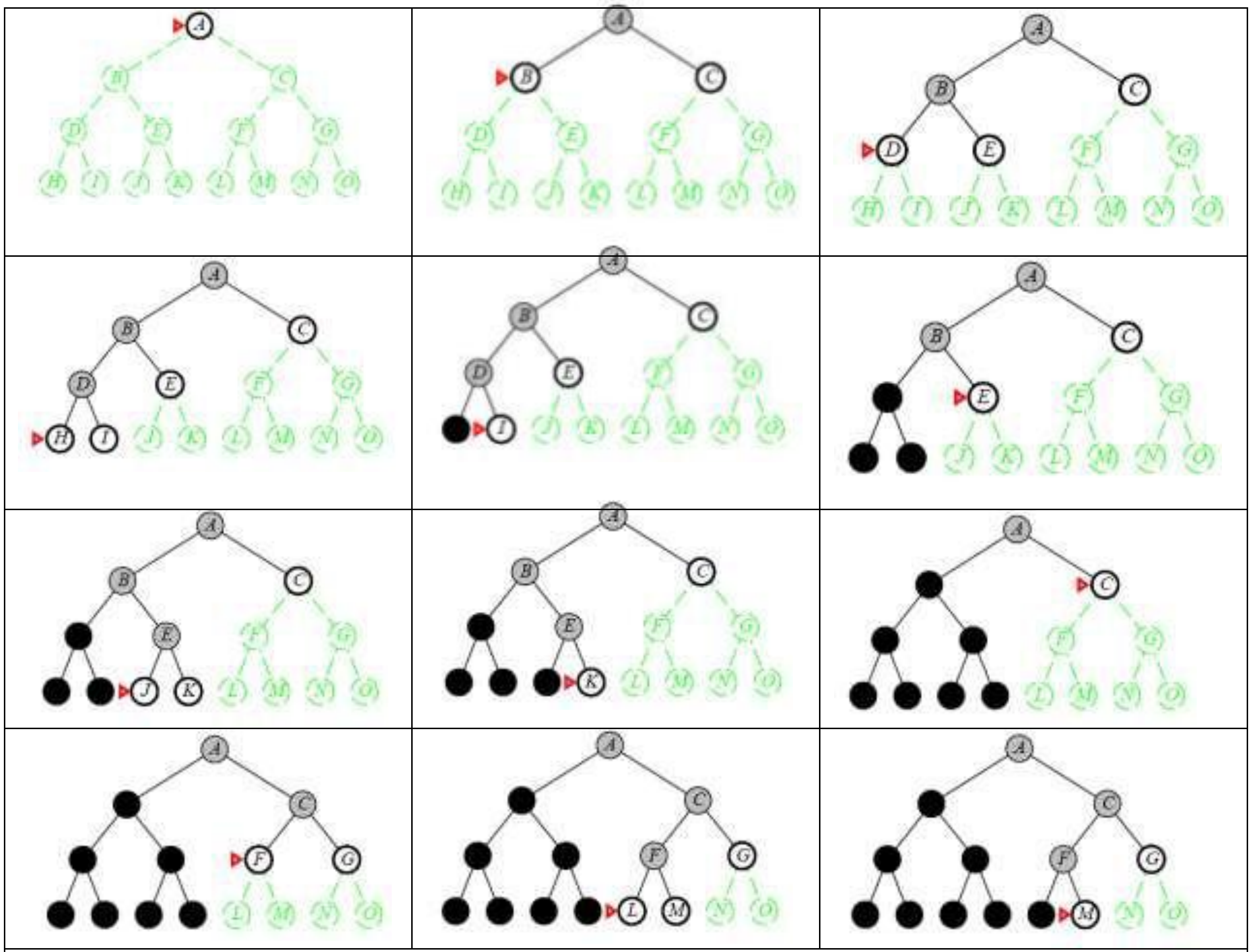


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

Depth-first-search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from the memory,as soon as its descendants have been fully explored(Refer Figure 1.31).
For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only bm + 1 nodes.

Using the same assumptions as Figure 2.11, and assuming that nodes at the same depth as the goal node have no successors, we find the depth-first-search would require 118 kilobytes instead of 10 petabytes, a factor of 10 billion times less space.

Drawback of Depth-first-search
The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

1.3.4.4 DEPTH-LIMITED-SEARCH

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.That is,nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem.
Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo
Sometimes,depth limits can be based on knowledge of the problem. For,example,on the map of Romania there are 20 cities. Therefore,we know that if there is a solution.,it must be of length 19 at the longest,So l = 10 is a possible choice. However,it oocan be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space,gives us a better depth limit.
Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth-limited-search is shown in Figure 1.32.
It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.
Depth-limited search = depth-first search with depth limit l,
returns cut off if any path is cut off by depth limit

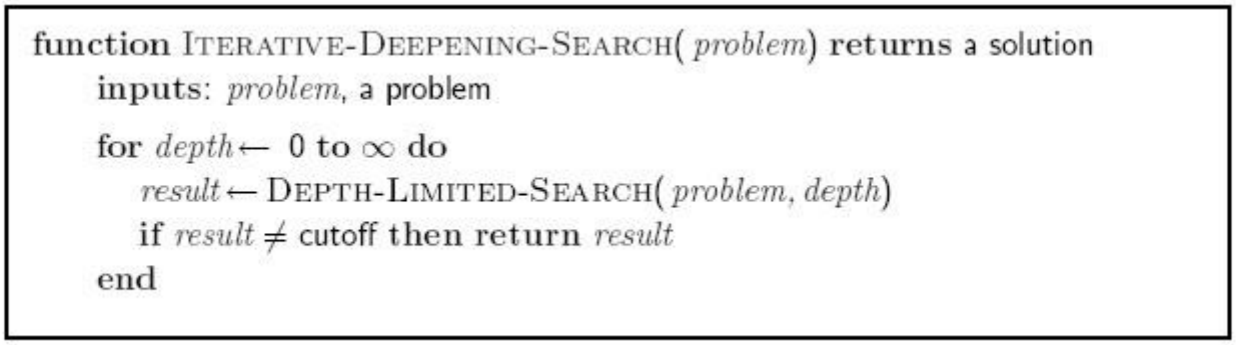
```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred? ← false
if Goal-Test(problem,State[node]) then return Solution(node)
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 not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

Figure 1.32 Recursive implementation of Depth-limited-search:

1.3.4.5 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-search,that finds the better 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. The algorithm is shown in Figure 2.14.
Iterative deepening combines the benefits of depth-first and breadth-first-search
Like depth-first-search,its memory requirements are modest;O(bd) to be precise.

Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.
Figure 2.15 shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

Figure 1.33 The **iterative deepening search algorithm** ,which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search resturns *failure*,meaning that no solution exists.

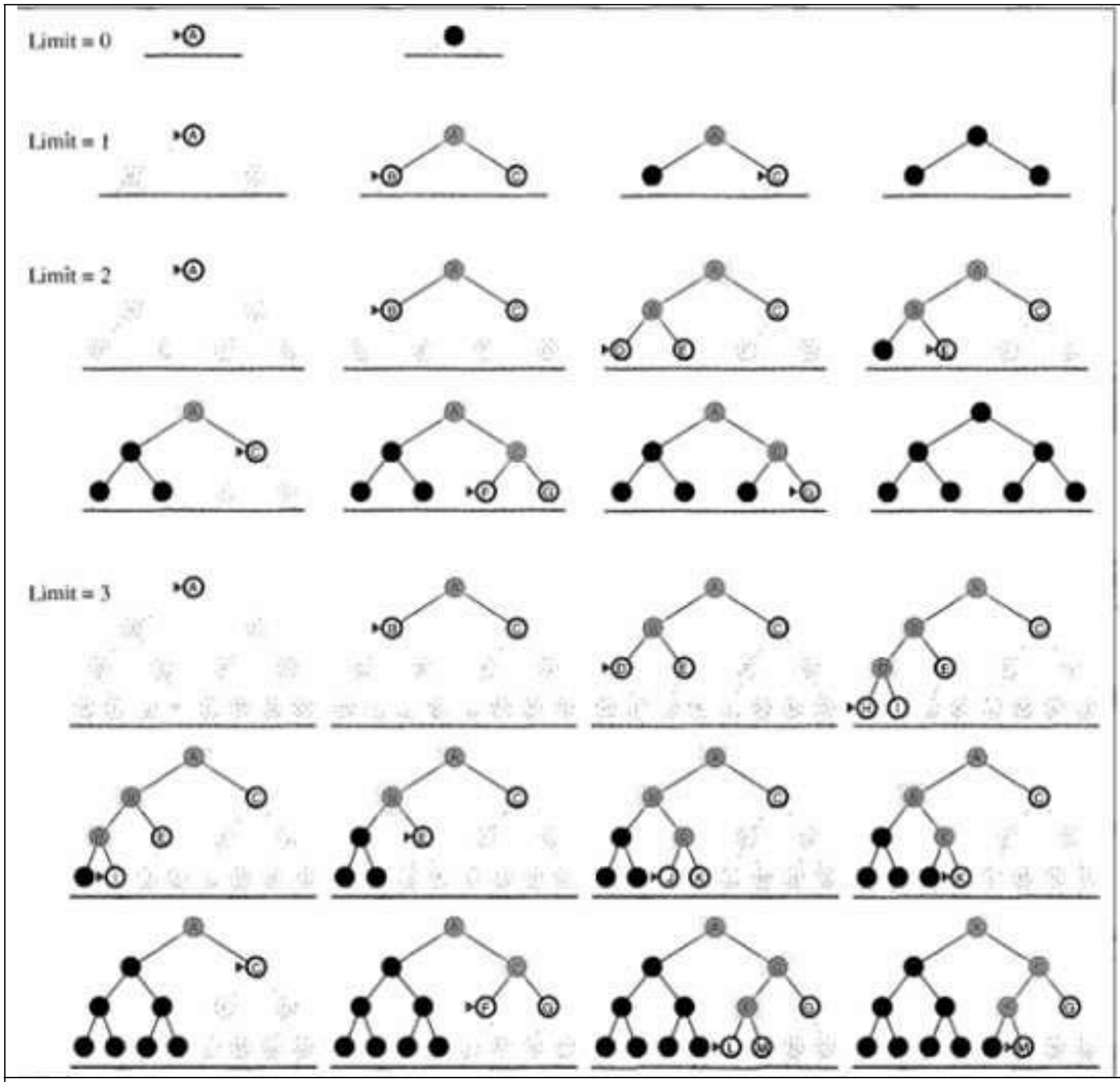


Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Iterative deepening search

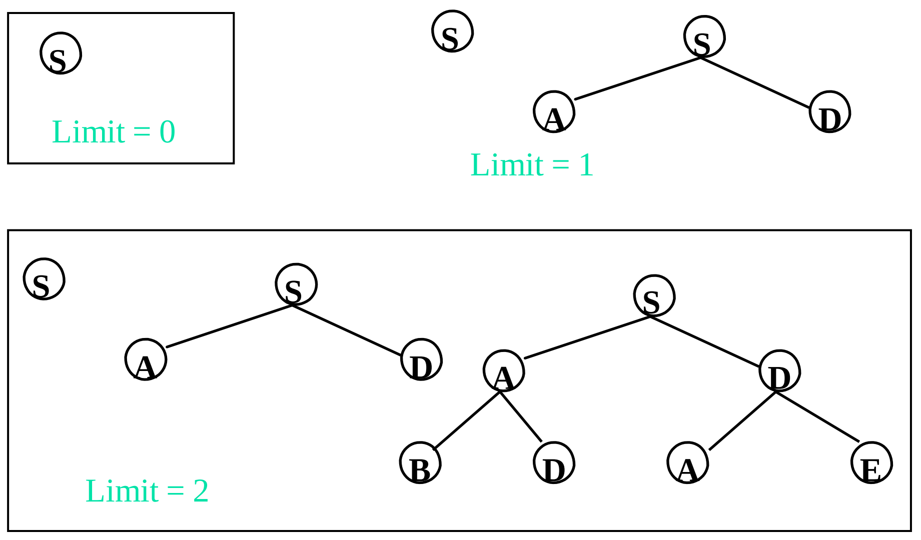
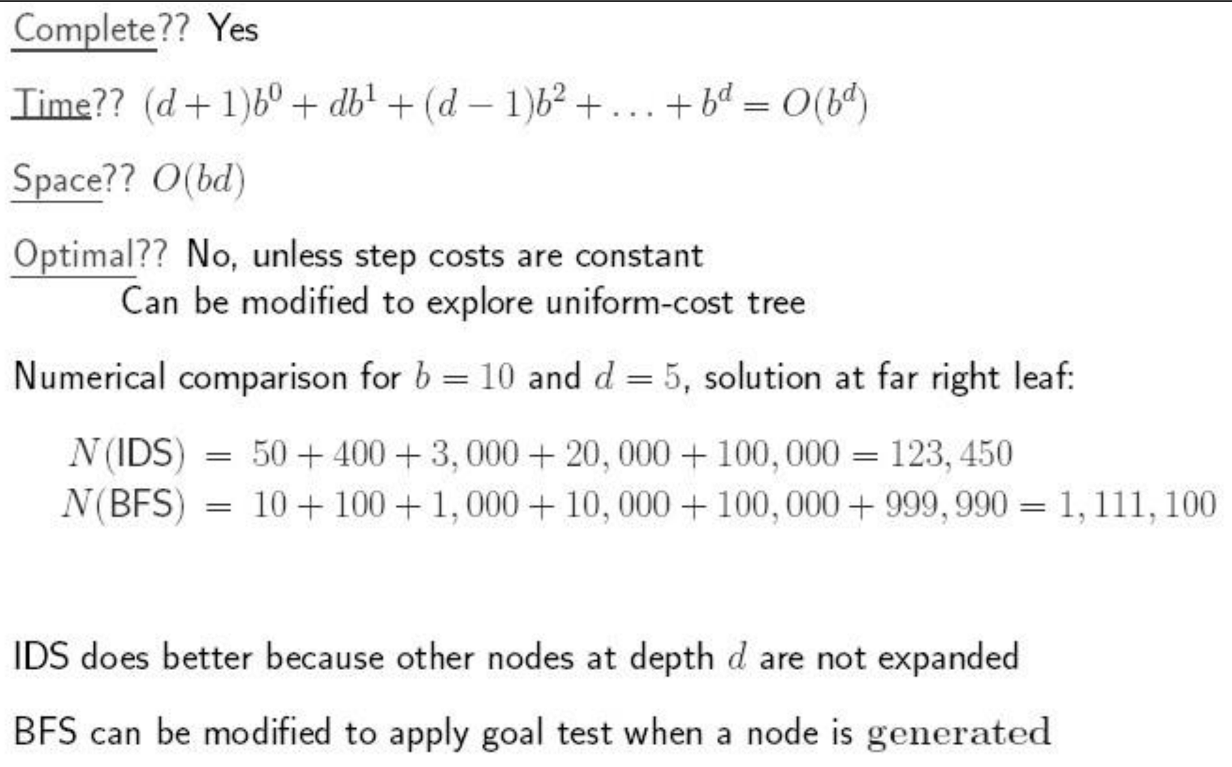


Figure 1.35

Iterative search is not as wasteful as it might seem

Properties of iterative deepening search



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

Figure 1.36

WEEK - 7

In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.