## 4.4 BEST FIRST SEARCH

This new method is a combination of both depth first search and breadth first search into a single method called Best First Search. It is defined as:

*"Best first search is little like hill climbing, in that it uses an evaluation function and always chooses the next node to be that with the best score. However, it is exhaustive, in that it should eventually try all possible paths. It uses an agenda as in breadth/depth first search, but instead of taking the first node off the agenda (and generating its successors) it will take the best node off, i.e. the node with the best score. The successors of the best node will be evaluated (i.e. have a score assigned to them) and added to the list."*

Best First Search also depends on the use of a heuristic to search most promising path to the goal node. Unlike hill climbing, however, Best First Search method retains all estimates computed for previously generated nodes and makes its selection based on the best among them all. Thus, at any point in the search process, Best First Search moves forward from the most promising of all the nodes generated so far. In doing so, it avoids the potential traps encountered in hill climbing. The heuristic function used here called an evaluation function is an indicator of how far the node is from the goal node. Goal nodes have an evaluation function value of zero. The key component of the heuristic function for the Best First Search can be denoted as $h(n)$. It is an estimated cost of the cheapest path from node $n$ to a goal node. If $n$ is the goal node then $h(n) = 0$.The Best First Search method can be describe in the following algorithm:

**Algorithm : Best First Search**

Suppose we have the following search tree:

Step 1: Place the starting node **s** on the queue.

Step 2: If the queue is empty, return failure and stop.

Step 3: If the first element on the queue is a goal node **g** , return success and stop. Otherwise,

Step 4: Remove the first element from the queue, expand it and compute the estimated goal distance for the each child. Place the children on the queue(at either end) and arrange all queue elements in ascending order corresponding to goal distance from the front of the queue.

Step 5: Return to step 2.

**Algorithm 4.4 : Best First Search Algorithm**

The Best First Search process is illustrated in figure 4.4, where number in the nodes may be regarded as estimates of the distance or cost to reach the goal node.

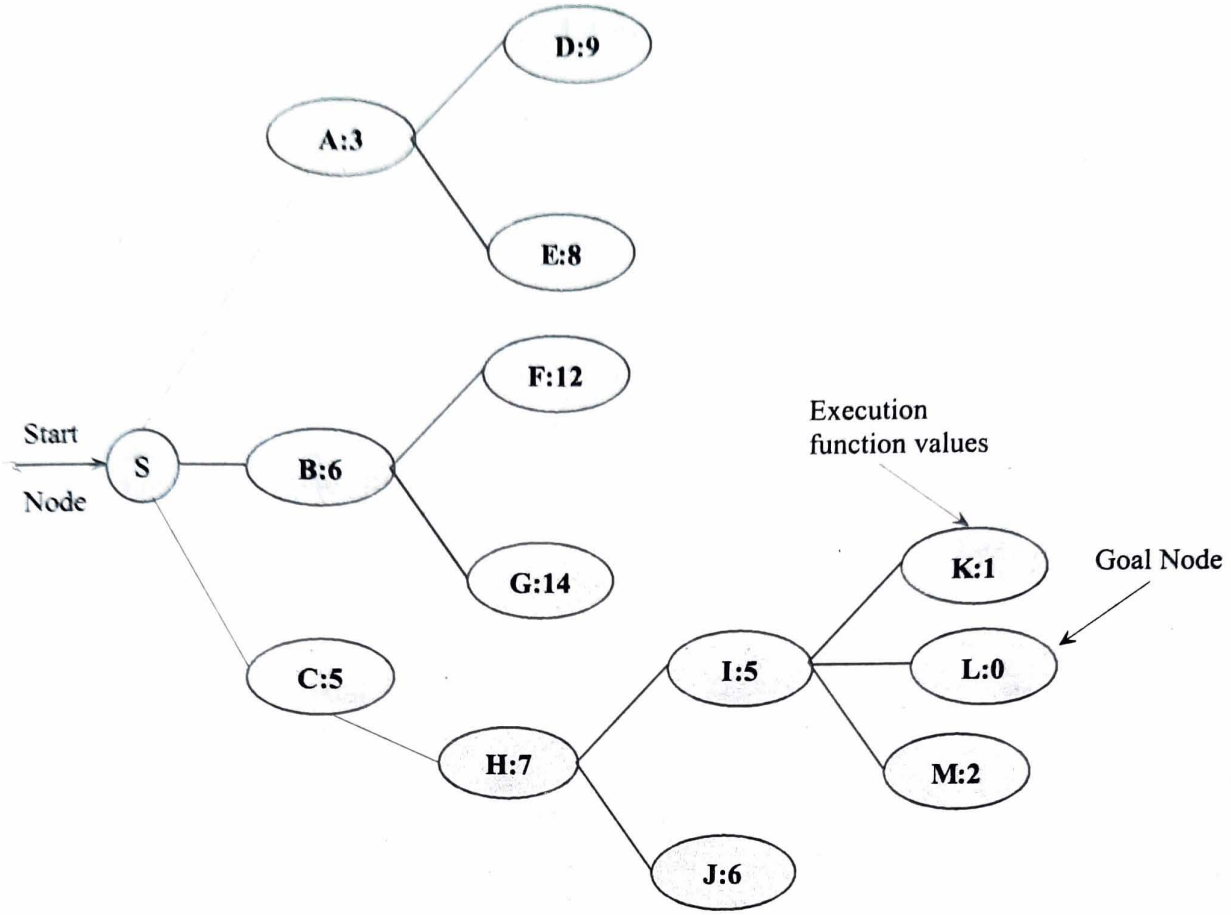



**Figure 4.4 : Search Process of Best First Search**

Here links between nodes illustrate possible successor states. A node label **B:6** means that node name is B and it has *n* estimated cost to solution of 6, (so a lower value is better).

Suppose our goal state is *L*. The goal state can be found with the Best First Search method by expanding the search graph. First, the start node S is expanded. It has three children A, B, C with the heuristic function value 3, 6 and 5 respectively. These values approximately indicate how far the are from the goal state. The child with minimum value namely A is chosen. The children of A are generated. They are D and E with the value 9 and 8. The search process has now four nodes to search for i.e. node D with value 9, node E with value 8, node B with value 6 and node C with value 5.Among them C has the minimal value which is expanded to give node H with value 7. At this point , the nodes available for search are (D:9), (E:8), (B:6) and (H 7). Among these nodes, B is minimum and hence B is expanded to give (F:12), (G:14).

Now the node available for search are (D:9), (E:8), (H:7) ,(F:12) and (G:14), out of which (H:7) is minimal and is expanded to give (I:5), (J:6). Nodes now available for expansion are (D:9), (E:8), (F:12), (G:14), (I:5), (J:6). Of these, the node with minimal value is (I:5) which is expanded to give the goal node.

The various steps are shown in the following table:

| Step | Node being expanded | Children | Available Nodes | Node chosen |
|---|---|---|---|---|
| 1. | S | (A : 3), (B : 6), (C : 5) | (A : 3), (B : 6), (C : 5) | (A : 3) |
| 2. | A | (D : 9),(E : 8) | (B : 6), (C : 5), (D : 9), (E : 8) | (C : 5) |
| 3. | C | (H : 7) | (B : 6), (D : 9) , (E : 8) , (H:7) | (B : 6) |
| 4. | B | (F : 12),(G : 14) | (D : 9) , (E : 8) , (H : 7), (F : 12), (G : 14) | (H : 7) |
| 5. | H | (I : 5),(J : 6) | (D : 9) , (E : 8), (F : 12),(G : 14) , (I : 5),(J : 6) | (I : 5) |
| 6. | I | (K : 1),(L : 0),(M: 2) | (D : 9) , (E : 8), (H : 7), (F : 12), (G : 14) , (J : 6), (K : 1),(L : 0),(M: 2) | Search stops as goal L is reached |

**Table 4.1 : Search Process of Best First Search Algorithm**

**Advantage:**

- The Best First Search resemble Depth First Search in the way it prefers to follow a single path all the way to goal, but will backup when it is a dead end.

**Disadvantage:**

- It suffers from the same defects as Depth First Search that it is not optimal, and it is incomplete (because it can go along an infinite path and never return to try other possibilities).

**Complexity:**

The worst case **time** and **space complexity** is $O(b^d)$ where $d$ is the maximum depth of the search space with good heuristic function, the complexity can be reduced substantially. The amount of reduction depends on the particular problem and the quality of the heuristic.

## 4.5 THE A* ALGORITHM

In its simplest form as described above, best first search is useful, but doesn't take into account the cost of the path so far when choosing which node to search from next. So, we may find a solution but it may be not a very good solution. There is a variant of best first search known as A* which attempts to find a solution by minimizing the total length or cost of the solution path. It combines advantages of breadth first search, where the shortest path is found first, with advantages of best first search, where the node that we guess is closest to the solution is explored next.

In the A* algorithm the score which is assigned to a node is a combination of the cost of the path so far and the estimated cost to solution. This is normally expressed as an evaluation

function $f$, which involves the sum of the values returned by two functions $g$ and $h$, $g$ returning the cost of the path (from initial state) to the node in question, and $h$ returning an estimate of the remaining cost to the goal state.

$$f(Node) = g(Node) + h(Node)$$

The previous heuristic methods offer good strategies but fail to describe how the shortest distance to a goal should be estimated. The A* algorithm is a specialization of Best First Search. It provides general guidelines with which to estimate goal distances for general search graphs.

For this type of problem, it is convenient to maintain two lists of node types designated as OPEN and CLOSED. Nodes on the OPEN list are nodes that have been generated but not yet expanded while nodes on the CLOSED list are nodes that have been expanded and whose children are available to the search program. The A* algorithm processed as follows:

**Algorithm: A* Search**

Step 1: Put the initial nodes on a list OPEN.

Step 2: If (OPEN is empty ) or (OPEN=GOAL), terminate search.

Step 3: Remove the first node from OPEN.Call this node **a**.

Step 4: If (a=GOAL), terminate search with success.

Step 5: Else if node **a** has successors, generate all of them. Estimate the fitness number of the successors by totalling the evaluation function value and the cost function value. Sort the list by fitness number.

Step 6: Name the new list as CLOSED.

Step 7: Replace OPEN with CLOSED

Step 8: GO to step 2.

**Algorithm 4.5 : A* Search Algorithm**

The **performance of A* Algorithm** can be evaluated with following important properties:

(a) ***Admissible Condition***: Algorithm A* is Admissible if it is guaranteed to return an optimal solution when one exists.

(b) ***Completeness Condition***: Algorithm A* is Completeness if it always terminates with a solution when one exists.

(c) ***Dominance Property***: Let $A_1$ and $A_2$ be Admissible algorithm with heuristic estimates function $h^*_1$ and $h^*_2$, respectively. $A_1$ is said to be more informed then $A_2$ whenever $h^*_1(n) > h^*_2(n)$ for all $n$. $A_1$ is also said to dominate $A_2$.

(d) ***Optimality Property:*** Algorithm A* is optimal over a class of algorithm if A dominates all members of the class.

A very interesting observation about this algorithm is that it is admissible that is, for any node $n$ on such path, $h'(n)$ is always less then or equal to $h(n)$.This is possible only when the evaluation function value never overestimates or underestimates (Rich & Knight ,1991), the distance of the node to the goal.

## 4.6 PROBLEM REDUCTION

Sometimes problems only seem hard to solve. A hard problem may be one that can be reduced to a number of simple problems and, when each of the simple problems is solved, then the hard problem has been solved. This is the basic intuition behind the method of problem reduction.

The typical problem that is used to illustrate problem reduction search is the *Tower of Hanoi* problem because this problem has a very elegant solution using this method. The story that is typically quoted to describe the Tower of Hanoi problem describes the specific problem faced by the priests of Brahmah as describe in the topic 3.5 of chapter 3. The gist of story is that 64 sizes ordered disks occupy one of 3 pegs and must be transferred to one of the other pegs. But, only one disk can be moved at a time; and a larger disk may never be placed on a smaller disk

Rather than dealing with the 64 disk problem faced by the priests, we will consider only three disks, the minimum required to make the problem mildly interesting and useful for our purpose here, mainly to illustrate problem reduction search. The problem involves moving from a state where the disks are stacked on one of the pegs and moving them so that they end up stacked on a different peg.

By contrast, problem reduction involves the use of operators which breakdown a complex problem possibly into several simpler, possibly independent, sub-problems each of which must be solved separately.

Any sub problem may itself be decomposed into sub problems. But, in order for this method to succeed, all sub problems must eventually terminate in primitive sub problems. A primitive sub problem is one which cannot be decomposed (i.e., there is no non-terminal that is applicable to the sub problem) and its solution is simple or direct. The terminal rules serve as recognizer of primitive sub problems.

In problem reduction search the problem space consists of an AND/OR graph of (partial) state pairs. These pairs are referred to as (sub) problems. The first element of the pair is the starting state of the (sub) problem and the second element of the pair is the goal state (sub) problem. By contrast, problem reduction involves the search of an AND/OR graph where

each node now represents only a part of the problem to be solved and each arc links a problem to a sub-problem arcs from a node which are bounded into groups. Each group of arcs from a node represents a particular way of decomposing the problem by one of the alternative operators which can be applied to that node. As each group is an alternative it can be regarded as the "or"part of the graph. Within each group, the various arcs link to nodes representing the sub-problems that need to be solved using that particular operator. Thus within a group the arcs are in an "AND" relation to each other ,because all the sub-problem must be solved. The problem for the system in this case is to find the interconnected paths that link the node which represents the starting state with all the nodes that represent the ultimate decomposition of the problem into its simplest sub-problems.

There are two types of generators: **non-terminal rules** and **terminal rules**. Non-terminal rules decompose a problem pair, *<s0, g0>* into an ANDed set of problem pairs *{<si, gi>, <sj, gj>, ...>*. The assumption is that the set of sub-problems are in some sense simpler problems than the problem itself. The set is referred to as an ANDed set because the assumption is that the solution of all of the sub-problems implies that the problem has been solved. Note all of the sub-problems must be solved in order to solve the parent problem. In other words we can say that problem reduction representation (PRR) problem is specified by a **3 Tuple (G, O, P).**

- **G** is a problem to be solved.
- **O** is a set of operators for decomposing problems into sub-problems through AND or OR decompositions.
- **P** is a set of primitive problem solution.
- An **AND** decomposition is solved when each of the sub-problems is solved.
- An **OR** decomposition is solved when at least one of the sub-problems is solved.
- A problem is unsolvable if it is neither a primitive problem nor can it be further decomposed.

### 4.6.1 AND OR Graphs

AND-OR graphs are useful for certain problems where the solution involves decomposing the problem into smaller problems and then these sub problems can be solved. Here the alternatives involve branches where some or all must be satisfied before we can progress. In AND OR graph:

- Nodes correspond to problems,
- Connectors correspond to area, and
- Connectors correspond to AND or OR decompositions.

An **AND Node** which can consist of many successor nodes, represents a given decomposition and all must be solved to find the goal. An **OR Node** which represents a choice between possible decompositions, indicates the various ways in which problem can be solved. For example, if goal is to learn to play a ***Frank Zappa guitar***, then he can transcribe from the CD OR buy the ***"Frank Zappa Guitar Book"*** **AND** read it. This is shown in figure 4.5.

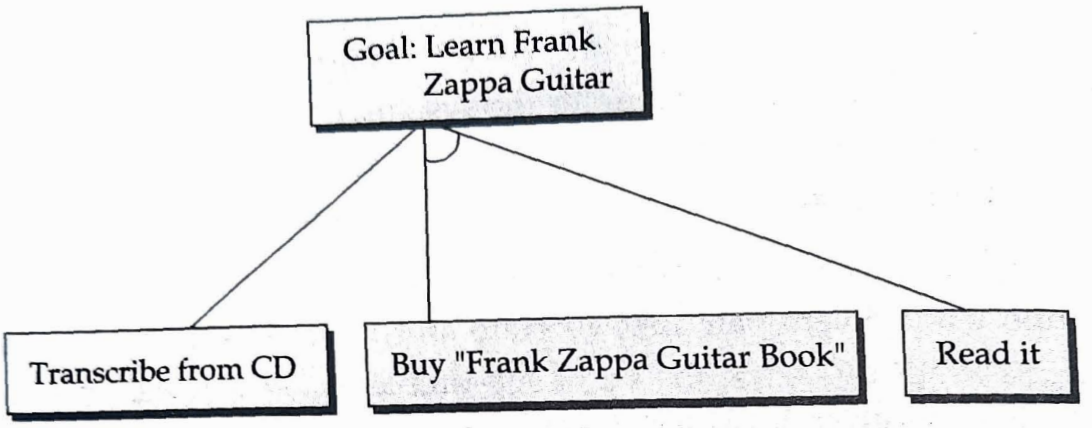



**Figure 4.5 : A simple AND-OR graph**

Best first search method can be used to solve AND-OR graphs, but it is not adequate for searching AND-OR graphs. Lets understand with the help of example, considering the graph of figure 4.6.

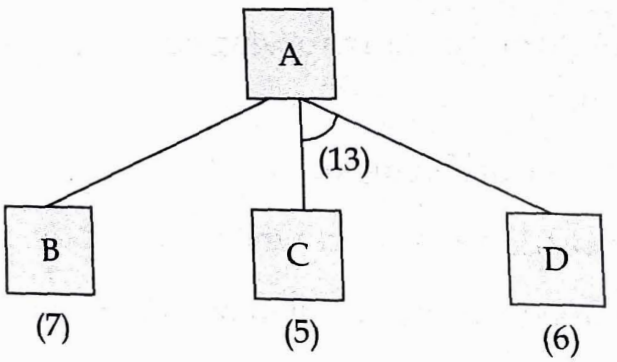



**Figure 4.6 : AND-OR graph**

Here in the graph, number adjacent to node is a value of heuristic function $f'$ and each arc with the successor has a cost of 1. The node with the lowest $f'$ value is expanded always because ;

***Heuristic function = min value***

So, node B is expanded because of lowest value. Node C is not expanded because it is AND node and requires D to be expanded so cost increases to 13 (5 + 6 + 2) then B's cost of 8 (7 + 1). The node B is explored as shown in figure 4.7.

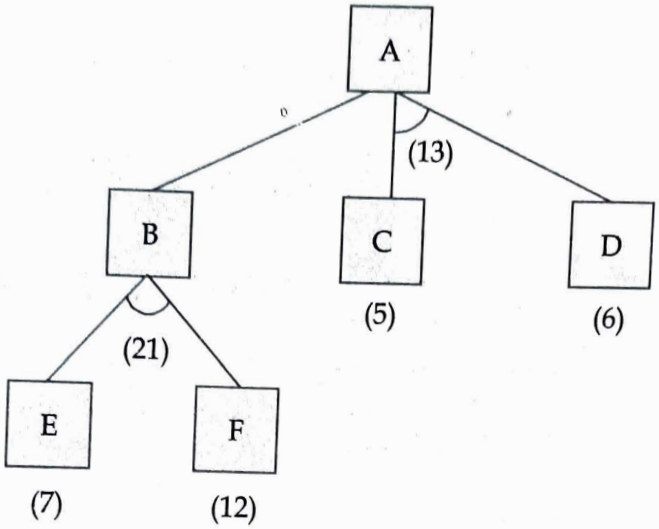



**Figure 4.7 : AND OR graph (exploring B)**

Now, it is seen that $f'$ value of B (21) is more than $f'$ value of A (13) so, C and D are explored. After exploring C and D the graph is shown in figure 4.8.

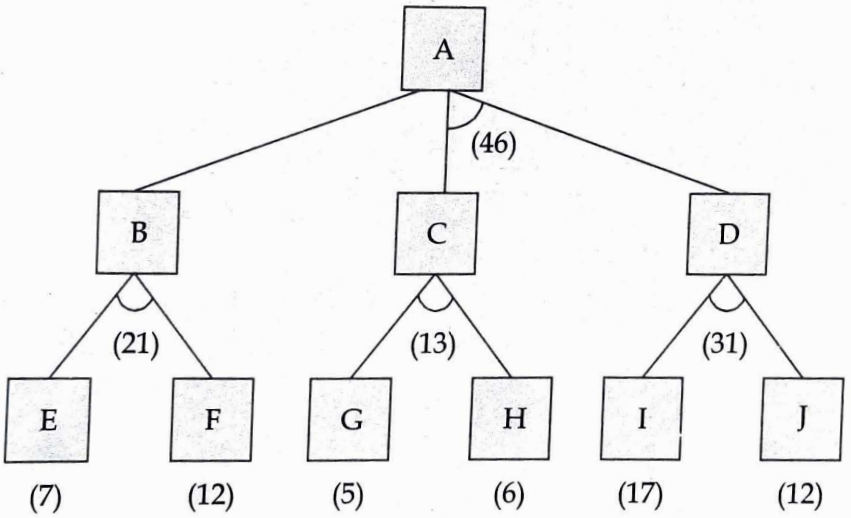



**Figure 4.8 : AND OR Graph (Exploring C and D)**

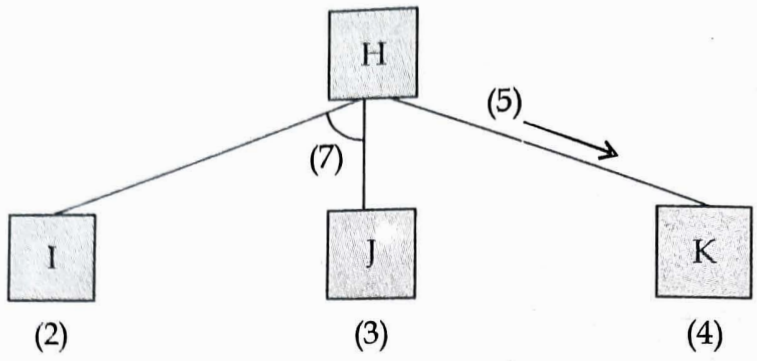
In this graph we can see that, exploring C and D are more costlier (46) than exploring E and F via A with a cost of 22 (22 + 1). We should not explore G any more and should examine E or F.

**Algorithm : Problem Reduction**

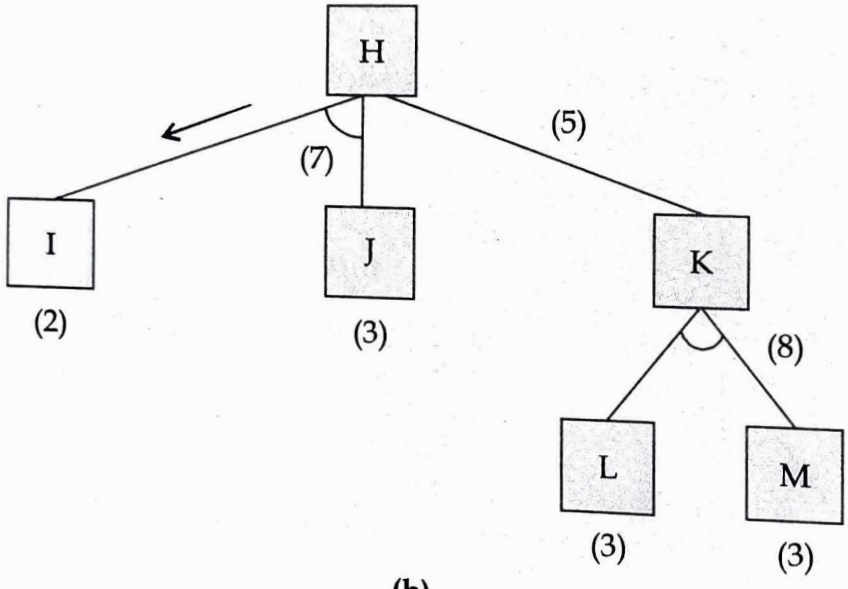
1. Initialize graph with root node.
2. Loop until starting node is solved or its cost exceeds **F** value.
   - Expand the nodes of best path until they are solved.
   - If no successors exist, assign **F** value to it otherwise, compute **f'** for all the successors. If **f'** is 0 label it **solved**.
   - Label node solved if all its successors are solved and propagate this change backward. This may cause best path to change.

**Algorithm 4.6 : Problem Reduction**

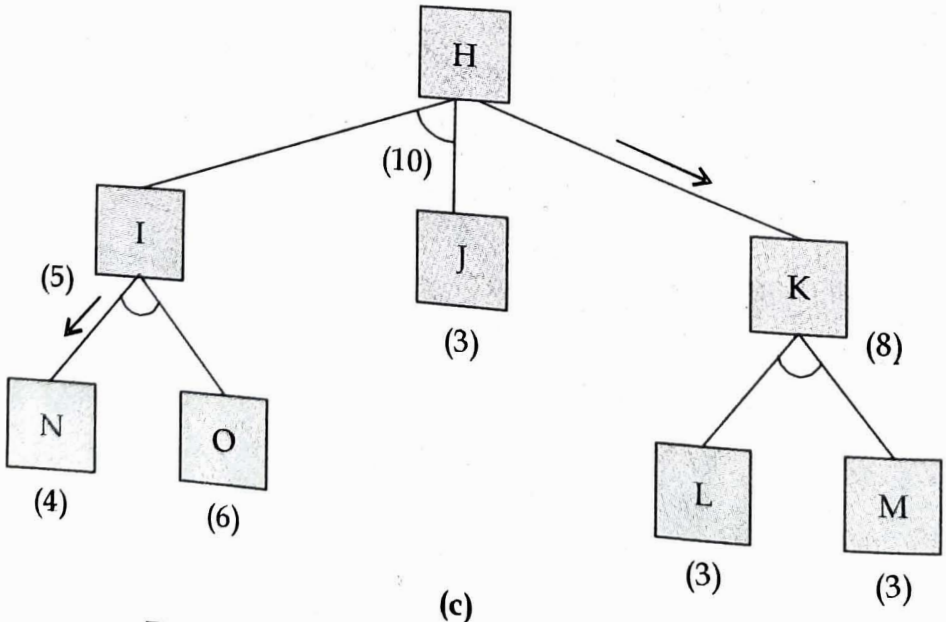
In algorithm 4.6, F is the futility value. If solution cost exceeds *F*, then search is aborted because it is too expensive. F is any numeric value that is cost effective for the solution. The above algorithm is illustrated with the help of example figure 4.9.




(a)




(b)




(c)

**Fig. 4.9 (a) - (c) : Problem Reduction Example**

(a) Node H is explored, yielding nodes I, J, K.

(b) Node K is explored because of lowest cost 5, producing L and M with AND arc. Update $f'$ value of K with 8. This makes path through I - J better than current path K. I - J are explored.

(c) Exploring I generates nodes, N and O. Propagating $f'$ value backwards, we find that path through K is again better than I.

This process either found a solution or dead end state (no solution).

### 4.6.2 AO* Algorithm

The depth first and breadth first search strategies given earlier for trees and graphs can easily be adapted for AND-OR trees. The main difference lies in the way termination conditions are determined.

As in the case of the A* algorithm, we use the OPEN list to hold nodes that have been generated but not explored and the CLOSED list to hold nodes that have been expanded (successor nodes that are available). The algorithm is a variation of the original given by *Nilsson (1971)*. It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for AND node solutions which require solutions to all successor nodes. A solution is found when the start node is labeled as solved. The steps of AO* algorithm are given below :

**Algorithm: AO***

Step 1: Initialize the graph to start node.

Step 2: Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved.

Step 3: Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only $f'$ for each of the successors.

Step 4: If $f'$ is 0 then mark the node as *SOLVED*.

Step 5: Change the value of $f'$ for the newly created node to reflect its successors by back propagation.

Step 6: Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.

Step 7: If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat step 2.

**Algorithm 4.7 : AO* Algorithm**

It can be shown that AO* will always find a minimum-cost solution tree if one exists.