K.D.K. COLLEGE OF ENGINEERING, NAGPUR

Introduction to Artificial Intelligence

UNIT: 2 SEARCH TECHNIQUES

By, Dr. Sanjay M. Malode

Assistant Professor,

Department of Artificial Intelligence and Data Science

Chapter 2: SEARCH TECHNIQUE:

Uninformed Search Strategies:

A problem determines the graph and the goal but not which path to select from the frontier. This is the job of a search strategy. A search strategy specifies which paths are selected from the frontier. Different strategies are obtained by modifying how the selection of paths in the frontier is implemented.

- **Uninformed search strategies**
  - Also known as "blind search," uninformed search strategies use no information about the likely "direction" of the goal node(s)
  - Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- **Informed search strategies**
  - Aka "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)
  - Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*

Evaluating search strategies

Completeness

Guarantees finding a solution whenever one exist

Time complexity

How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded

Space complexity

How much space is used by the algorithm? Usually measured in terms of the maximum size of the "nodes" list during the search

Optimality/Admissibility

If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

Depth-First Search

The first strategy is **depth-first search**. In depth-first search, the frontier acts like a last-in first-out **stack**. The elements are added to the stack one at a time. The one selected and taken off the frontier at any time is the last element that was added.

Algorithm :

If the initial state is a goal state, quit and return success

Otherwise, do the following until success or failure is signalled:

- Generate a successor, E, of initial state.
- If there are no more successors, signal failure.
- Call Depth-First Search, with E as the initial state.
- success is returned, signal success. Otherwise continue in this loop.

Properties of DFS

If it is known that the solution path will be long, DFS will not spend time searching a large number of "shallow" states in the graph.

However, DFS may "lost" deep in a graph, missing short paths to a goal, or even stuck in an infinite loop.

Advantages of DFS:

DFS requires less memory since only the nodes on the current path are stored.

By chance, DFS may find a solution without examining much of the search space at all.

Breadth-First Search

In **breadth-first search** the frontier is implemented as a FIFO (first-in, first-out) queue. Thus, the path that is selected from the frontier is the one that was added earliest.

This approach implies that the paths from the start node are generated in order of the number of arcs in the path. One of the paths with the fewest arcs is selected at each stage.

Breadth-first search is useful when

- space is not a problem;
- you want to find the solution containing the fewest arcs;

- few solutions may exist, and at least one has a short path length; and
- Infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

Algorithm:

1) Create a variable called NODE-LIST and set it to initial state.
2) Until a goal state is found or NODE-LIST is empty do.
   - Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.
   - For each way that each rule can match the state described in E do:
     - Apply the rule to generate a new state.
     - If the new state is a goal state, quit and return this state.
     - Otherwise, add the new state to the end of NODE-LIST .

Properties of BFS

Because it always examines all nodes at level n before proceeding to level n+1, BFS always finds the shortest path to a goal.

In a problem has a simple solution, the solution will be found.

Unfortunately, if the states have a high average number of children, it may use all the memory before it find a solution

Advantages of BFS:

BFS will not get trapped exploring a blind alley.

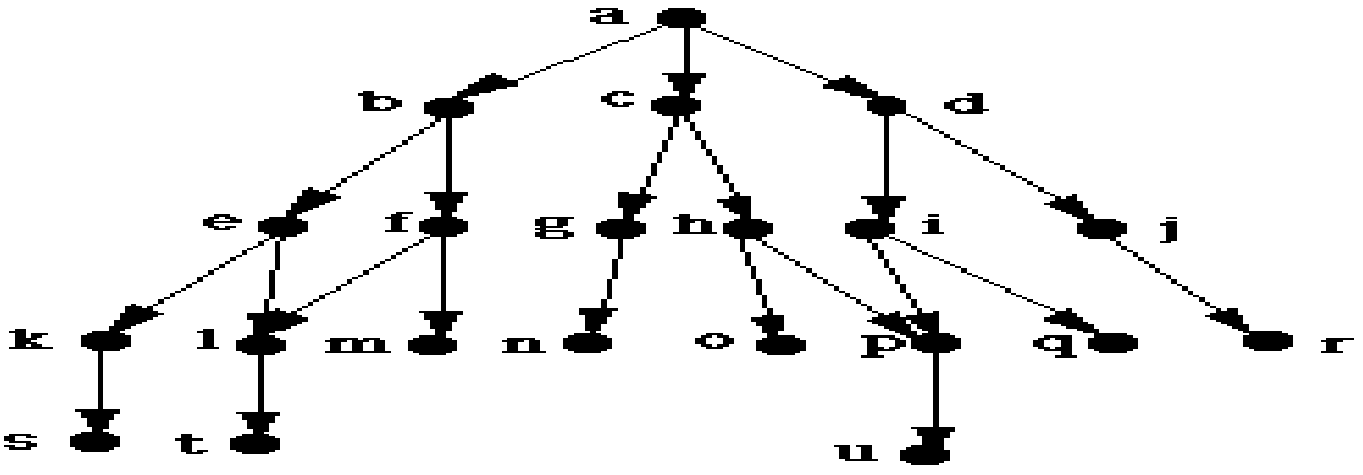
If there is a solution, BFS is guaranteed to find it.

If there are multiple solutions, then a minimal solution will be found.

Example: Consider the tree-shaped graph. Suppose the start node is the root of the tree (the node at the top) and the nodes are ordered from left to right so that the leftmost neighbour is added to the stack last.

Implementing the frontier as a stack results in paths being pursued in a depth-first manner - searching one path to its completion before trying an alternative path. This method is said to involve **backtracking**: The algorithm selects a first alternative at each node, and it *backtracks* to the next alternative when it has pursued all of the paths from the first selection. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.

This algorithm does not specify the order in which the neighbours are added to the stack that represents the frontier. The efficiency of the algorithm is sensitive to this ordering.



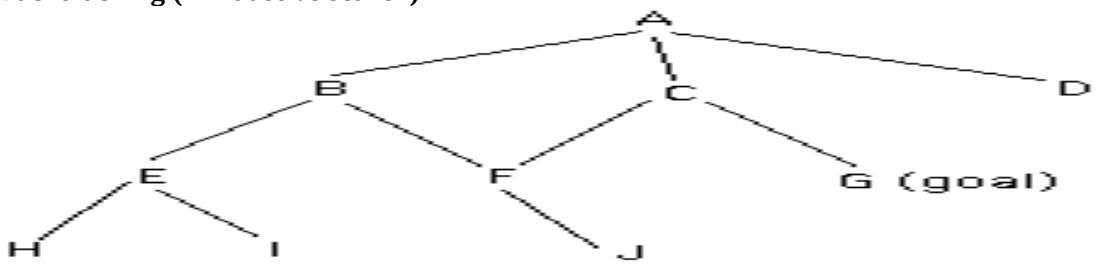
Depth-first search

A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R

Breadth-first search

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U

Backtracking (Exhaustive search)



Exhaustive search, try each path in order, until find goal. The following shows the "Depth-first search" version of exhaustive search. Start at A. Search state space systematically until find goal. When multiple children, go down 1st child. If fails, back to parent, down 2nd child, and so on. Note there are multiple paths to F. If F has already been found to be a dead-end when we went there from B, the algorithm should not go there a second time (from C).

Heuristic Search:

A Heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness.

Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions. They are bad to the extent that they may miss points of interest to particular individuals. On the average they improve the quality of the paths that are explored.

Using Heuristics, we can hope to get good (though possibly non-optimal) solutions to hard problems such as a TSP in non exponential time.

A *heuristic* is a method that might not always find the best solution but is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency. Useful in solving tough problems which could not be solved any other way. Solutions take an infinite time or very long time to compute. The *classic* example of heuristic search methods is the travelling salesman problem.

Nearest Neighbour Heuristic:

It works by selecting locally superior alternative at each step.

Applying to TSP:

Arbitrarily select a starting city.

Select the next city, look at all cities not yet visited and select the one closest to the current city. Go to next step.

Repeat step 2 until all cities have been visited.

This procedure executes in time proportional to N^2 and is possible to prove an upper bound on the error it incurs. This provides reassurance that one is not paying too high a price in accuracy for speed.

Heuristic function:

This is a function that maps from problem state descriptions to measures of interest, usually represented as numbers, which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well designed heuristic functions can play an important part in efficiently guiding a search process toward a solution.

Introduction Hill climbing

Artificial Intelligence search algorithms Search techniques are general problem-solving methods. When there is a formulated search problem, a set of states, a set of operators, an initial state, and a goal criterion we can use search techniques to solve the problem (Pearl & Korf, 1987)

Hill Climbing Solutions include:

Backtracking, making big jumps (to handle plateaus or poor local maxima) and applying multiple rules before testing (helps with ridges).

Hill-Climbing Methodology:

- Construct a sub-optimal solution that meets the constraints of the problem

● Take the solution and make an improvement upon it
● Repeatedly improve the solution until no more improvements are necessary/possible

Here the generate and test method is augmented by an heuristic function which measures the closeness of the current state to the goal state.

1. Evaluate the initial state if it is goal state quit otherwise current state is initial state.
2. Select a new operator for this state and generate a new state.
3. Evaluate the new state
    - If it is closer to goal state than current state make it current state
    - If it is no better ignore
4. If the current state is goal state or no new operators available, quit. Otherwise repeat from 2.

In the case of the four cubes a suitable heuristic is the sum of the number of different colors on each of the four sides, and the goal state is 16 four on each side. The set of rules is simply choose a cube and rotate the cube through 90 degrees. The starting arrangement can either be specified or is at random.

- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity.
- Often useful when combined with other methods, getting it started right in the right general neighbourhood.

Disadvantages:

- Backtrack to some earlier node and try going in a different direction.
- Make a big jump to try to get in a new section.
- Moving in several directions at once

Steepest-Ascent Hill Climbing (Gradient Search)

Algorithm:

1. Evaluate the initial state.
2. Loop until a solution is found or a complete iteration produces no change to current state:
    - SUCC = a state such that any possible successor of the current state will be better than SUCC (the worst state).
    - For each operator that applies to the current state, evaluate the new state:
        goal → quit
        better than SUCC → set SUCC to this state
    - SUCC is better than the current state → set the current state to SUCC.

SIMULATED ANNEALING

This is a variation on hill climbing and the idea is to include a general survey of the scene to avoid climbing false foot hills.

The whole space is explored initially and this avoids the danger of being caught on a plateau or ridge and makes the procedure less sensitive to the starting point. There are two additional changes; we go for minimisation rather than creating maxima and we use the term objective function rather than heuristic. It becomes clear that we are valley descending rather than hill climbing. The title comes from the metallurgical process of heating metals and then letting them cool until they reach a minimal energy steady final state. The probability that the metal will jump to a higher energy level is given by $p = \exp^{e-\Delta E/kT}$ where *k* is Boltzmann's constant. The rate at which the system is cooled is called the annealing schedule. $\Delta E$is called the change in the value of the objective function and *kT* is called *T* a type of temperature. An example of a problem suitable for such an algorithm is the travelling salesman. The SIMULATED ANNEALING algorithm is based upon the physical process which occurs in metallurgy where metals are heated to high temperatures and are then cooled. The rate of cooling clearly affects the finished product. If the rate of cooling is fast, such as when the metal is quenched in a large tank of water the structure at high temperatures persists at low temperature and large crystal structures exist, which in this case is equivalent to a local maximum. On the other hand if the rate of cooling is slow as in an air based method then a more uniform crystalline structure exist equivalent to a global maximum. The probability of making a large uphill move is lower than a small

move and the probability of making large moves decreases with temperature. Downward moves are allowed at any time.

1. Evaluate the initial state.
2. If it is goal state then quit otherwise make the current state this initial state and proceed.
3. Make variable *BEST_STATE* to current state
4. Set temperature, *T*, according to the annealing schedule
5. Repeat

ΔE-- difference between the values of current and new states

1. If this new state is goal state Then quit
2. Otherwise compare with the current state
3. If better set *BEST_STATE* to the value of this state and make the current the new state
4. If it is not better then make it the current state with probability *p*'. This involves generating a random number in the range 0 to 1 and comparing it with a half, if it is less than a half do nothing and if it is greater than a half accept this state as the next current is a half.
5. Revise T in the annealing schedule dependent on number of nodes in tree

Until a solution is found or no more new operators

6. Return *BEST_STATE* as the answer

Best First Search

A combination of depth first and breadth first searches.

Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends. The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better then this previously unexpanded node and branch is not forgotten and the search method reverts to the descendants of the first choice and proceeds, backtracking as it were.

This process is very similar to steepest ascent, but in hill climbing once a move is chosen and the others rejected the others are never reconsidered whilst in best first they are saved to enable revisits if an impasse occurs on the apparent best path. Also the best available state is selected in best first even its value is worse than the value of the node just explored whereas in hill climbing the progress stops if there are no better successor nodes. The best first search algorithm will involve an OR graph which avoids the problem of node duplication and assumes that each node has a parent link to give the best node from which it came and a link to all its successors. In this way if a better node is found this path can be propagated down to the successors. This method of using an OR graph requires 2 lists of nodes

OPEN is a priority queue of nodes that have been evaluated by the heuristic function but which have not yet been expanded into successors. The most promising nodes are at the front. CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

Heuristics In order to find the most promising nodes a heuristic function is needed called *f*' where *f*' is an approximation to *f* and is made up of two parts *g* and *h*' where *g* is the cost of going from the initial state to the current node; *g* is considered simply in this context to be the number of arcs traversed each of which is treated as being of unit weight. *h*' is an estimate of the initial cost of getting from the current node to the goal state. The function *f*' is the approximate value or estimate of getting from the initial state to the goal state. Both *g* and *h*' are positive valued variables. Best First The Best First algorithm is a simplified form of the *A** algorithm. From *A** we note that $f' = g+h'$ where *g* is a measure of the time taken to go from the initial node to the current node and *h*' is an estimate of the time taken to solution from the current node. Thus *f*' is an estimate of how long it takes to go from the initial node to the solution. As an aid we take the time to go from one node to the next to be a constant at 1.

Best First Search Algorithm:

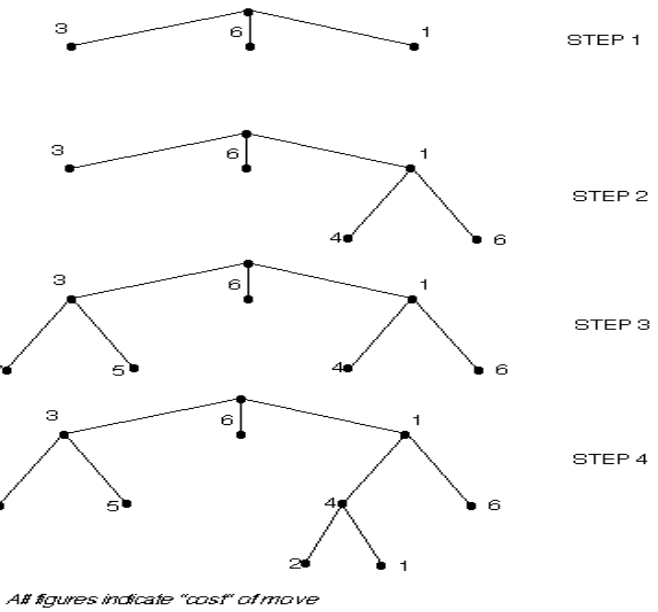
1. Start with OPEN holding the initial state
2. Pick the best node on OPEN
3. Generate its successors
4. For each successor Do
   - If it has not been generated before evaluate it add it to OPEN and record its parent
   - If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes
5. If a goal is found or no more nodes left in OPEN, quit, else return to 2.

Best-First Search

Algorithm

1. OPEN = {initial state}.
2. Loop until a goal is found or there are no nodes left in OPEN:
   - Pick the best node in OPEN
   - Generate its successors
   - For each successor:
     new → evaluate it, add it to OPEN, record its parent
     generated before → change parent, update successors

33



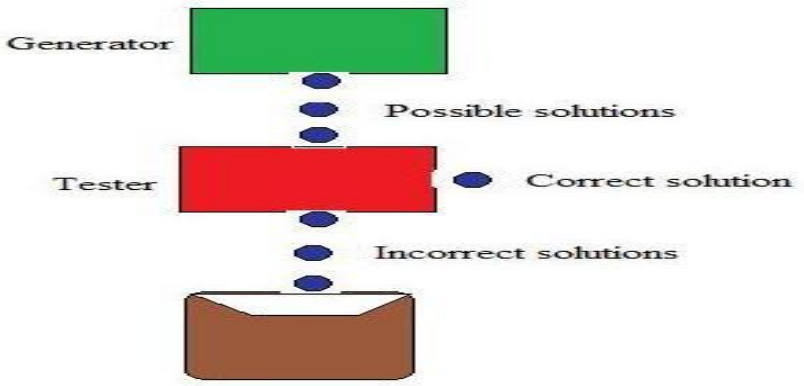
Generate-And-Test Algorithm

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exist a solution.

Algorithm: Generate-And-Test

1. Generate a possible solution.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.



Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

Systematic Generate-And-Test

While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

Generate-And-Test And Planning

Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted. An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures. Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

CONSTRAINT SATISFACTION:

Many AI problems can be viewed as problems of constraint satisfaction.

Cryptarithmetic puzzle: SEND + MORE =MONEY

As compared with a straightforard search procedure, viewing a problem as one of constraint satisfaction can reduce substantially the amount of search. Operates in a space of constraint sets.

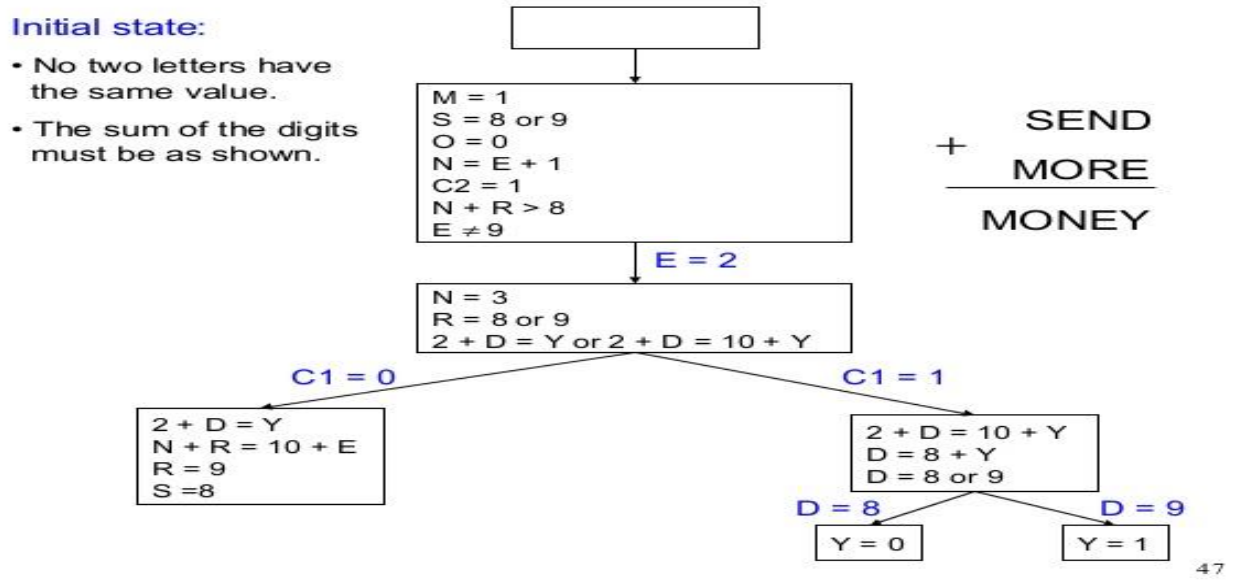
Initial state contains the original constraints given in the problem. A goal state is any state that has been constrained "enough".

Constraint Satisfaction Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Constraint Satisfaction Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.



The *A** Algorithm

Best first search is a simplified *A**.

1. Start with *OPEN* holding the initial nodes.
2. Pick the *BEST* node on *OPEN* such that $f = g + h'$ is minimal.
3. If *BEST* is goal node quit and return the path from initial to *BEST* Otherwise
4. Remove *BEST* from *OPEN* and all of *BEST*'s children, labelling each with its path from initial node.

PROBLEM REDUCTION (AND - OR graphs - AO * Algorithm)

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A* algorithm cannot search AND - OR graphs efficiently. This can be understood from the give figure.

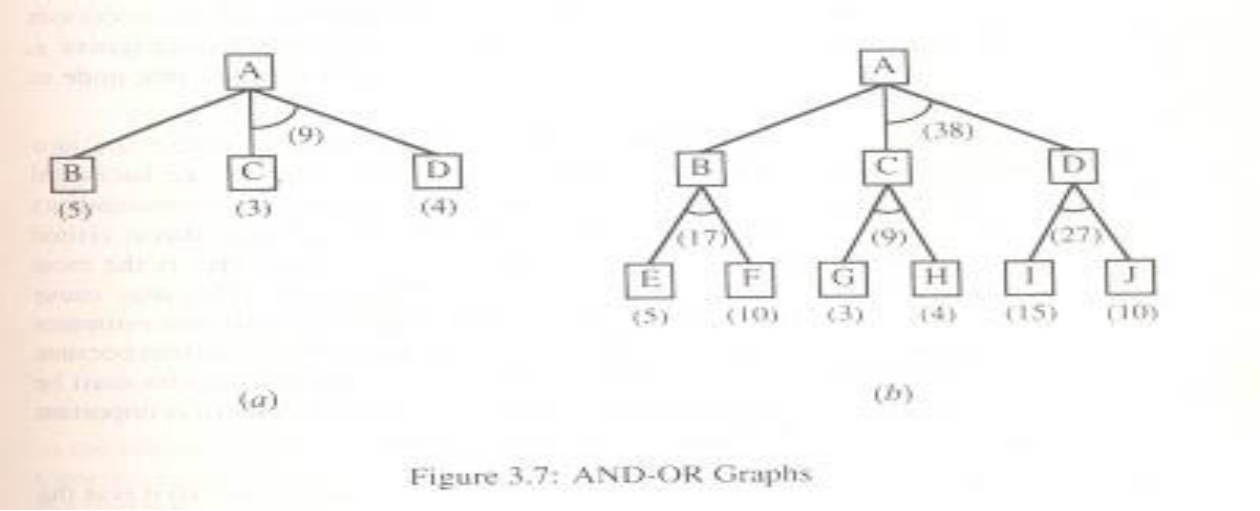


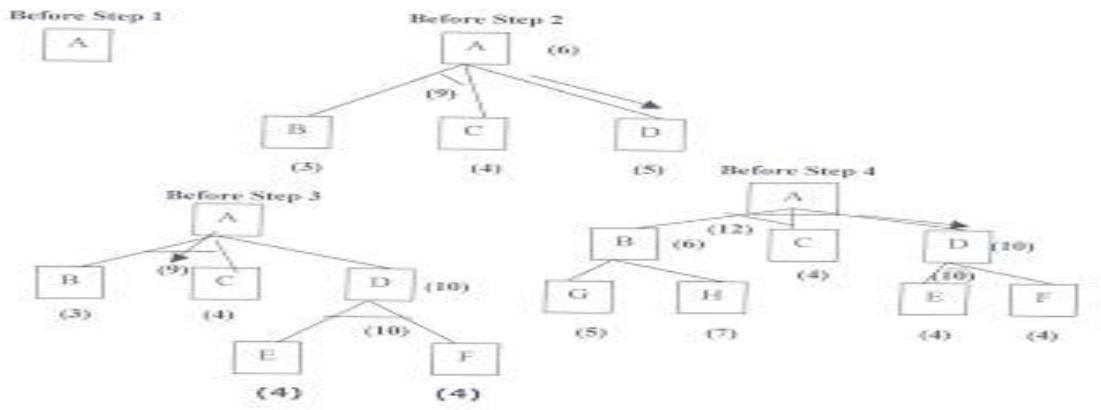
Figure 3.7: AND-OR Graphs

In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.
3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:



Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better . it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expensive to be practical.

For representing above graphs AO* algorithm is as follows:

AO* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).
2. Until INIT is labelled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

(II) Generate the successors of NODE. If there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancestor of NODE do the following:

(a) add SUCCESSOR to graph G

(b) if successor is not a terminal node, mark it solved and assign zero to its h ' value.

(c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following . let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;

(a) select a node from S call if CURRENT and remove it from S.

(b)compute h' of each of the arcs emerging from CURRENT , Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labelled SOLVED.

(e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status mustbe propagate backwards up the graph. Hence all the ancestors of CURRENT are added to S.

AO* Search Procedure:

1. Place the start node on open.
2. Using the search tree, compute the most promising solution tree TP.
3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.
4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.
5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open ,with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.
7. Go back to step(2)

Note: AO* will always find minimum cost solution.