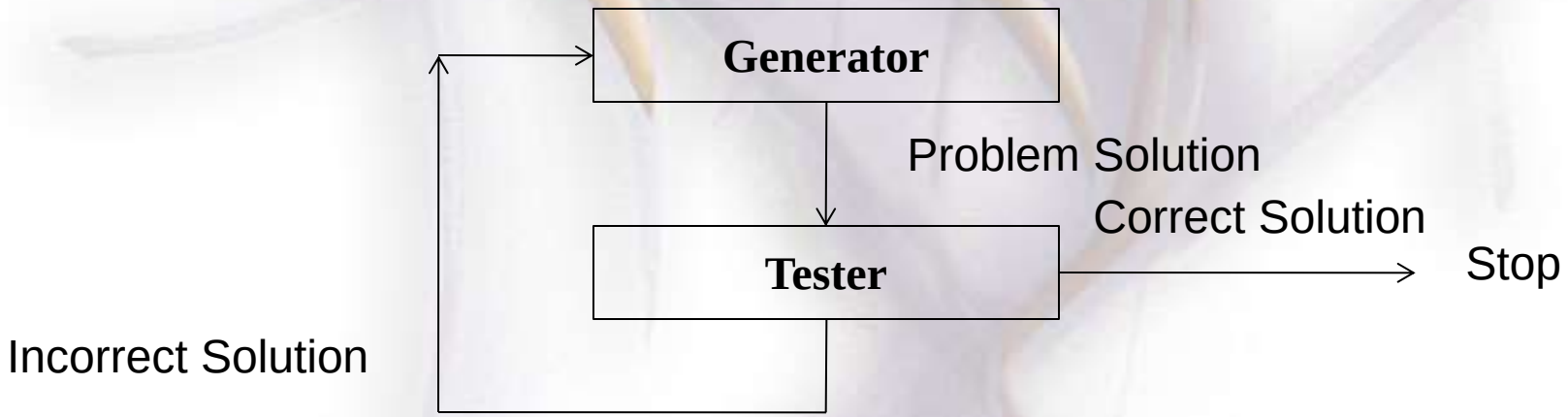
Heuristic Search Techniques

The Search Techniques

- In the previous chapter we have seen the concept of uninformed or blind search strategy.
- Now we will see more efficient search strategy, the informed search strategy.
- Informed search strategy is the search strategy that uses problem-specific knowledge beyond the definition of the problem.
- In general, methods followed in informed search is best first search.

Generate-And-Test

- **Generate-And-Test is a search algorithm which uses depth-first search techniques.**
- **It assure to find solution in systematic way.**
- **It generate complete solution and then the testing is done.**
- **A heuristic is needed so that the search is improved.**



Generate-And-Test

- **Following is the algorithm for Generate-And-Test:**

1) **Generate a possible solution which can either be a point in the problem space or a path from the initial state.**
2) **Test to see if this possible solution is a real(actual) solution by computing the state reached with the set of goal states.**
3) **If it is real solution then return the solution otherwise repeat from state 1.**

- **Generate and Test is acceptable for simple problem whereas it is inefficient for problem with large spaces.**

Best First Search Techniques

1) Search will start at root node.
2) The node to be expanded next is selected on the basis of an evaluation function f(n).
3) The node having value lowest value for f(n) is selected first. This lowest value of f(n) indicates that goal is nearest from this node (that is f(n) indicates distance from current node to goal node).

- BFS can be implemented using priority queue where fringe will be stored. The node in fringe will be stored in priority queue with increasing value of f(n) i.e. ascending order of f(n). The high priority will be given to node which has low f(n) value.

Best First Search Techniques

- As name says “best first” then, we would always expected to have optimal solution. But in general BFS indicates that choose the node that appears to be best according to the evaluation function.
- Hence the optimality based on ‘best-ness’ of evaluation function.

Best First Search Techniques

- **ALGORITHM FOR BEST FIRST SEARCH**

1. **Use two ordered lists OPEN and Close**
2. **Start with the initial node ‘n1’ and put it on the ordered list.**
3. **Create a list CLOSED. This is initially an empty list.**
4. **If OPEN is empty then exit with failure.**
5. **Select first node on OPEN. Remove it from OPEN and put it on CLOSED. Call this node n.**
6. **If ‘n’ is the goal node exit. The solution is obtained by tracing a path backward along the arcs in the tree from ‘n’ to ‘n1’.**

Best First Search Techniques

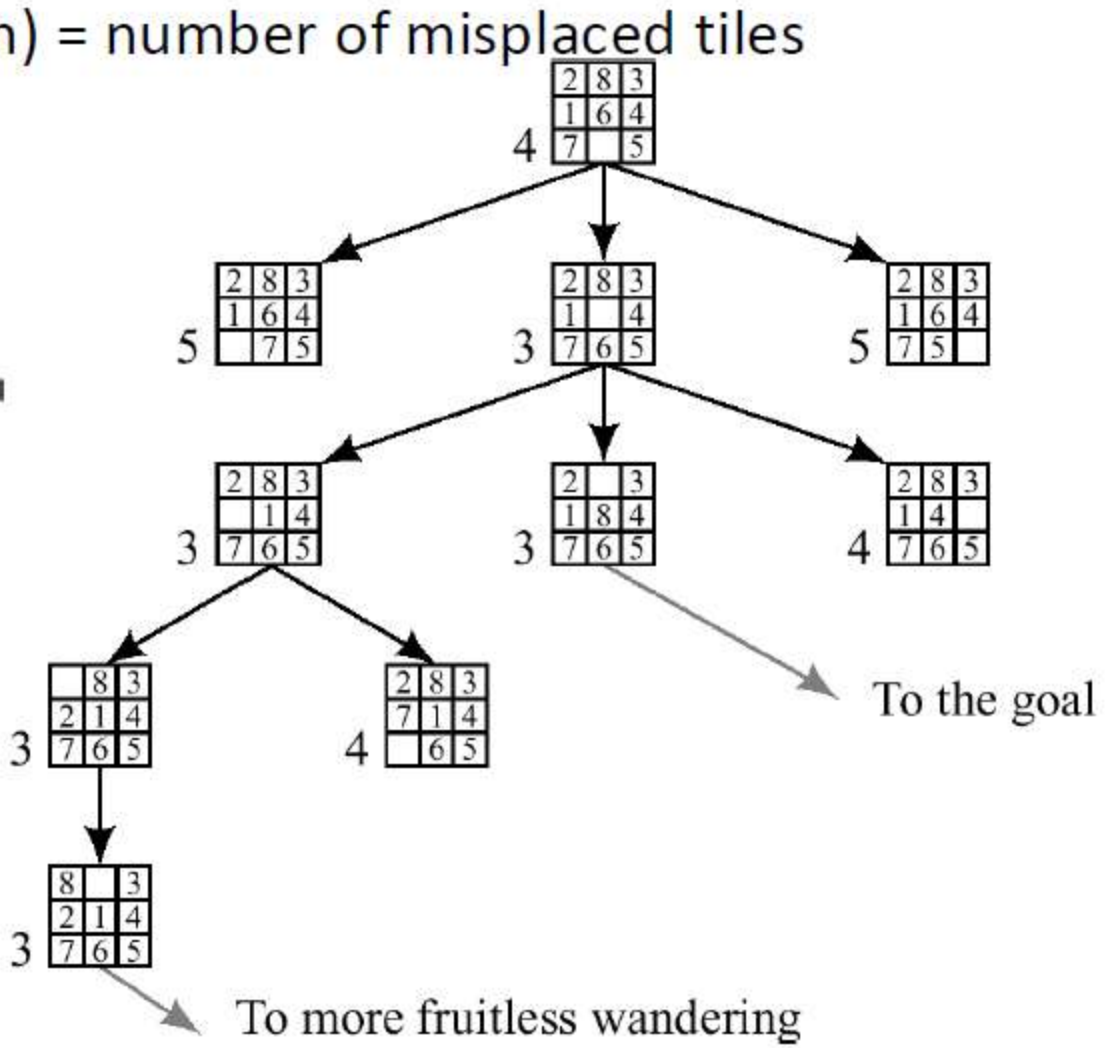
7. Expand node ‘n’. This will generate successors. Let the set of successor generated, be S. Create arcs from ‘n’ to each member of S.
8. Reorder the list OPEN, according to the heuristic and go back to step 4.

- Consider the following 8-puzzle problem – Here the heuristic used could be “number of tiles not in correct position” (i.e. number of tiles in misplaced).
- In this solution the convention used is, that smaller value of the heuristic function ‘f’ leads earlier to the goal state.

Best First Search Techniques

Figure 8.1

Start and Goal Configurations for the Eight-Puzzle



Heuristic Function

- There are many different algorithms which employ the concept of best first search.
- The main difference between all the algorithms is that they have different evaluation functions. The central component of these algorithms is heuristic function – h(n) which is defined as, h(n) = estimate cost of the cheapest path from node ‘n’ to a goal node.
- Heuristic function is key component of best first search. It is denoted by h(n) and h(n) = the shortest and cheapest path from the initial node to goal node.
- One can give additional knowledge about the problem to the heuristic function.

Heuristic Function

- A heuristic function guide the search in an efficient way.
- A heuristic function h(n) consider a node as input but it rely only on the state at that node.
- h(n) = 0, if n is goal state.

A* Algorithm

- A* is most popular form of Best First Search.
- A* evaluates node based on two functions namely
  1. g(n) – The cost to reach the node ‘n’.
  2. H(n) – the cost to reach the goal node from node ‘n’.
- These two function’s costs are combined into one, to evaluate a node. New function f(n) is derived as

  f(n) = g(n) + h(n) that is,

  f(n) = Estimated cost of the cheapest solution through ‘n’

A* Algorithm

- **ALGORITHM**

1) **The algorithm maintains two sets**
   - **OPEN list : The OPEN list keeps track of those nodes that need to be examined.**
   - **CLOSED list: The CLOSED list keeps track of nodes that have already been examined.**
2) **Initially, the OPEN list contains just the initial node, and CLOSED list is empty. Each node n maintains the following : g(n), h(n), f(n) as described below.**
3) **Each node also maintains a pointer to its parent, so that the best solution, if found, can be retrieved. A* has a main loop that repeatedly gets the node, call it 'n', with the lowest f(n) value from the open list. If 'n' is the goal node, then we are done, and the solution is given by backtracking from 'n'. Otherhrwise, 'n' is removed from the OPEN list and added to the CLOSED list. Next all the possible successor nodes of 'n' are generated.**

A* Algorithm

4) For each successor node ‘n’, if it is already in the CLOSED list and the copy there has an equal or lower ‘f’ estimate, and then we can safely discard the newly generated ‘n’ and move on. Similarly, if ‘n’ is already in the OPEN list and the copy there has an equal or lower f estimate, we can discard the newly generated n and move on.

5) If no better version of ‘n’ exists on either the CLOSED or OPEN lists, we remove the inferior copies from the two lists and set ‘n’ as the parent of ‘n’. We also have to calculate the cost estimates for n as follows:

   Set g(n) which is g(n) plus the cost of getting from n to n;

   Set h(n) is the heuristic estimate of getting from n to the goal node;

   Set f(n) is g(n) + h(n)

6) Lastly, add ‘n’ to the OPEN list and return to the beginning of the main loop.

A* Algorithm

Consider the situation shown in Fig. 3.4. Assume that the cost of all arcs is 1. Initially, all nodes except A are on *OPEN* (although the Fig. shows the situation two steps later, after B and E have been expanded). For each node, f' is indicated as the sum of h' and g. In this example, node B has the lowest f', 4, so it is expanded first. Suppose it has only one successor E, which also appears to be three moves away from a goal. Now $f'(E)$ is 5, the same as $f'(C)$. Suppose we resolve this in favor of the path we are currently following. Then we will expand E next. Suppose it too has a single successor F, also judged to be three moves from a goal. We are clearly using up moves and making no progress. But $f'(F) = 6$, which is greater than $f'(C)$. So we will expand C next. Thus we see that by underestimating $h'(B)$ we have wasted some effort. But eventually we discover that B was farther away than we thought and we go back and try another path.

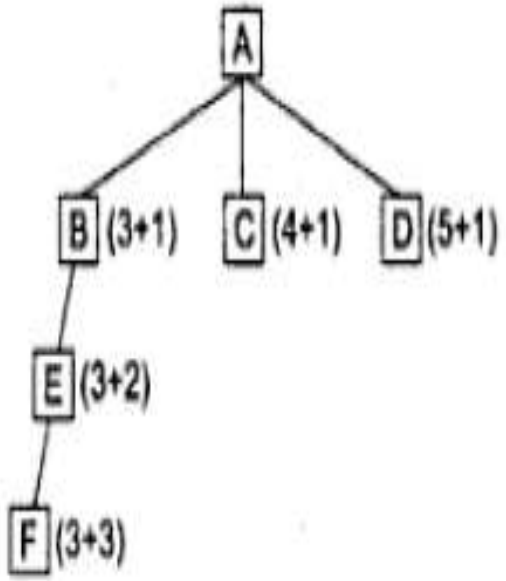


Fig. 3.4 *h' Underestimates h*

A* Algorithm

Now consider the situation shown in Fig. 3.5. Again we expand B on the first step. On the second step we again expand E. At the next step we expand F, and finally we generate G, for a solution path of length 4. But suppose there is a direct path from D to a solution, giving a path of length 2. We will never find it. By overestimating $h'(D)$ we make D look so bad that we may find some other, worse solution without ever expanding D. In general, if h' might overestimate h, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are longer than the best solution. An interesting question is, "Of what practical significance is the theorem that if *h!* never overestimates h then A* is admissible?" The answer is, "almost none," because, for most real problems, the only way to guarantee that *hi* never overestimates h is to set it to zero. But then we are back to breadth-first search, which is admissible but not efficient. But there is a corollary to this theorem that is very useful. We can state it loosely as follows:

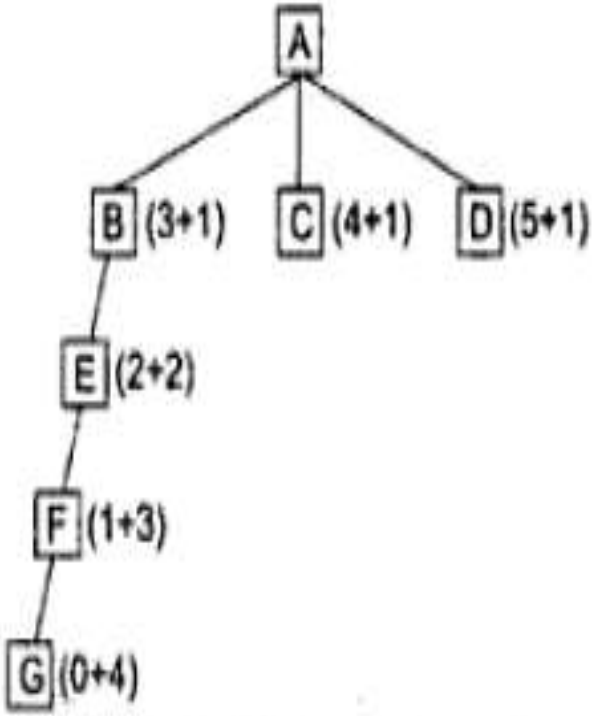


Fig. 3.5 *h' Overestimates h*

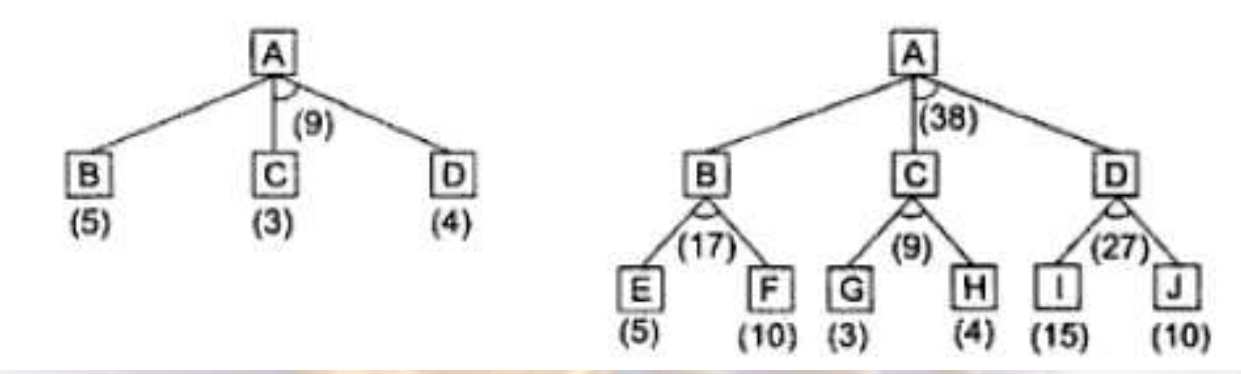
AO* Algorithm / AND-OR Graphs

- When a problem can be divided in a set of sub problems, where each sub problem can be solved separately and a combination of this will be a solution.
- AND-OR graph and AND-OR trees are used to representing the solution.

  - The decomposition of the problem or problem reduction generates AND arc.
  - One AND arc may point to any number of successor nodes.
- All these must be solved so that the arc will give rise to many arcs, indicating several possible solutions. Hence the graph is known as AND-OR instead of AND.
- AO* is a best-first algorithm for solving problems represented as a cyclic AND/OR graph problems.
- An algorithm to find a solution in an AND-OR graph must handle AND area appropriately.

AO* Algorithm / AND-OR Graphs

- To see why our best-first search is not adequate for searching AND-OR graph, consider following figure.



- The top node A has been expended, producing two arcs, one leading to B and one leading to C and D. The number at each node represents the value of f' at that node.
- We assume, for simplicity, that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with multiple successor has a cost of 1 for each of its component.
- If we just look at the node and choose for expansion the one with the lowest f' value, we must select C.

AO* Algorithm / AND-OR Graphs

- But using the information now available, it would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C + D + 2) compared to the cost of 6that we get by going through B.
- The problem is that the choice of which node to expand next must depend not only on the f' value of that node but also on whether that node is part of the current best path from the initial node.

THE COMPARATIVE STUDY ON A* and AO*

- Unlike A* algorithm which used two list OPEN and CLOSED, the AO* algorithm uses a single structure G.
- G represents the part of search graph generated do far.
- Each node in G points down to its immediate successor and upto its immediate predecessor, and also has with it the value of 'h' cost of a path from itself to a set of solution node.

AO* Algorithm / AND-OR Graphs

- The cost of getting from the start nodes to the current node ‘g’ is not stored as in A* algorithm. This is not possible to compute a single such value since there may be many paths to the same state.
- AO* algorithm serves as the estimate of goodness if a node.
- A* algorithm can not search AND-OR graph efficiently.
- AO* will always find minimum cost solution.

AO* ALGORITHM

1. Initialize the graph to start node.
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved.
3. Pick any of these node and expand it and if it has no successors call this value as, FUTILITY, otherwise calculate only f’ for each of the successors.
4. If f’ is 0 then mark the node as SOLVED.

AO* Algorithm / AND-OR Graphs

5. Change the value of f’ for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as SOLVED then mark the parent node as SOLVED.
7. If starting node is SOLVED or value greater than FUTILITY, stop, else repeat from 2.