CHAPTER 2

Unit II

Problem Solving

Syllabus

2.1 Problem Solving Agent, Formulating Problems, Example Problems

2.2 Uninformed Search Methods : Depth Limited Search, Depth First Iterative Deepening (DFID), Informed Search Method : A* Search

2.3 Optimization Problems : Hill climbing Search, Simulated annealing, Genetic algorithm

Search is an indivisible part of intelligence. An intelligent agent is the one who can search and select the most appropriate action in the given situation, among the available set of actions. When we play any game like chess, cards, tic-tac-toe, etc.; we know that we have multiple options for next move, but the intelligent one who searches for the correct move will definitely win the game. In case of travelling salesman problem, medical diagnosis system or any expert system; all they required to do is to carry out search which will produce the optimal path, the shortest path with minimum cost and efforts. Hence, this chapter focuses on the searching techniques used in AI applications. Those are known as un-informed and informed search techniques.

2.1 Solving Problems by Searching

- Now let us see how searching play a vital role in solving AI problems. Given a problem, we can generate all the possible states it can have in real time, including start state and end state. To generate solution for the same is nothing but searching a path from start state to end state.
- Problem solving agent is the one who finds the goal state from start state in optimal way by following the shortest path, thereby saving the memory and time. It's supposed to maximize its performance by fulfilling all the performance measures.
- Searching techniques can be used in game playing like **Tic-Tac-Toe** or navigation problems like **Travelling Salesman Problem.**
- First, we will understand the representation of given problem so that appropriate searching techniques can be applied to solve the problem.

2.2 Formulating Problems

MU - Dec. 12

Q. Explain how you will formulate search problem. **(Dec. 12, 3 Marks)**

- Given a goal to achieve; problem formulation is the process of deciding what states to be considered and what actions to be taken to achieve the goal. This is the first step to be taken by any problem solving agent.
- **State space :** The state space of a problem is the set of all states reachable from the initial state by executing any sequence of actions. State is representation of all possible outcomes.
- The state space specifies the relation among various problem states thereby, forming a directed network or graph in which the nodes are states and the links between nodes represent actions.

- **State Space Search :** Searching in a given space of states pertaining to a problem under consideration is called a state space search.
- **Path :** A path is a sequence of states connected by a sequence of actions, in a given state space.

2.2.1 Components of Problems Formulation

MU - May 15

Q. Explain steps in problem formulation with example. (May 15, 10 Marks)

Problem can be defined formally using five components as follows :

1.	Initial state	2.	Actions
3.	Successor function	4.	Goal test
5.	Path cost		

1. **Initial state :** The initial state is the one in which the agent starts in.
2. **Actions :** It is the set of actions that can be executed or applicable in all possible states. A description of what each action does; the formal name for this is the transition model.
3. **Successor function :** It is a function that returns a state on executing an action on the current state.
4. **Goal test :** It is a test to determine whether the current state is a goal state. In some problems the goal test can be carried out just by comparing current state with the defined goal state, called as **explicit goal test**. Whereas, in some of the problems, state cannot be defined explicitly but needs to be generated by carrying out some computations, it is called as **implicit goal test**.

   For example : In Tic-Tac-Toe game making diagonal or vertical or horizontal combination declares the winning state which can be compared explicitly; but in the case of chess game, the goal state cannot be predefined but it's a scenario called as "Checkmate", which has to be evaluated implicitly.
5. **Path cost :** It is simply the cost associated with each step to be taken to reach to the goal state. To determine the cost to reach to each state, there is a cost function, which is chosen by the problem solving agent.

   Problem solution : A well-defined problem with specification of initial state, goal test, successor function, and path cost. It can be represented as a data structure and used to implement a program which can search for the goal state. A solution to a problem is a sequence of actions chosen by the problem solving agent that leads from the initial state to a goal state. Solution quality is measured by the path cost function.

   Optimal solution : An optimal solution is the solution with least path cost among all solutions.

A general sequence followed by a simple problem solving agent is, first it formulates the problem with the goal to be achieved, then it searches for a sequence of actions that would solve the problem, and then executes the actions one at a time.

2.2.2 Example Problems

MU - Dec. 12

Q. Formulate 8-puzzle problem. (Dec. 12, 3 Marks)

1. Example of 8-puzzle problem

- Fig. 2.2.1 depicts a typical scenario of 8-puzzle problem. It has a 3×3 board with tiles having 1 through 8 numbers on it. There is a blank tile which can be moved forward, backward, to left and to right. The aim is to arrange all the tiles in the goal state form by moving the blank tile minimum number of times.

1	2	3
4	8	–
7	6	5

Initial State

1	2	3
4	5	6
7	8	–

Goal State

Fig. 2.2.1 : A scenario of 8-Puzzle Problem

- This problem can be formulated as follows :

States : States can represented by a 3 x 3 matrix data structure with blank denoted by 0.

1. **Initial state :** {{1, 2, 3},{4, 8, 0},{7, 6, 5}}
2. **Actions :** The blank space can move in Left, Right, Up and Down directions specifying the actions.
3. **Successor function :** If we apply "Down" operator to the start state in Fig. 2.2.1, the resulting state has the 5 and the blank switching their positions.
4. **Goal test :** {{1, 2, 3},{4, 5, 6},{7, 8, 0}}
5. **Path cost :** Number of steps to reach to the final state.

Solution :

{{1, 2, 3},{4, 8, 0},{7, 6, 5}}⟶ {{1, 2, 3},{4, 8, 5},{7, 6, 0}} →{{1, 2, 3},
{4, 8, 5},{7, 0, 6}} →{{1, 2, 3},{4, 0, 5},{7, 8, 6}}⟶{{1, 2, 3},{4, 5, 0},{7, 8, 6}} →{{1, 2, 3},{4, 5, 6},{7, 8, 0}}
Path cost = 5 steps

2. Example of missionaries and cannibals problem

- The problem statement as discussed in the previous section. Let's formulate the problem first.
- **States :** In this problem, state can be data structure having triplet (i, j, k) representing the number of missionaries, cannibals, and canoes on the left bank of the river respectively.

1. **Initial state :** It is (3, 3, 1), as all missionaries, cannibals and canoes are on the left bank of the river.
2. **Actions :** Take x number of missionaries and y number of cannibals
3. **Successor function :** If we take one missionary, one cannibal the other side of the river will have two missionaries and two cannibals left.
4. **Goal test :** Reached state (0, 0, 0)
5. **Path cost :** Number of crossings to attain the goal state.

Solution :

The sequence of actions within the path :

(3,3,1) → (2,2,0)→(3,2,1) →(3,0,0) →(3,1,1) →(1,1,0) →(2,2,1) →(0,2,0) →(0,3,1) →(0,1,0) → (0,2,1) →(0,0,0)

Cost = 11 crossings

3. Vacuum-cleaner problem

States : In vacuum cleaner problem, state can be represented as [<block>, clean] or [<block>, dirty]. The agent can be in one of the two blocks which can be either clean or dirty. Hence there are total 8 states in the vacuum cleaner world.

1. **Initial State :** Any state can be considered as initial state. For example, [A, dirty]
2. **Actions :** The possible actions for the vacuum cleaner machine are left, right, absorb, idle.

3. **Successor function :** Fig. 2.2.2 indicating all possible states with actions and the next state.

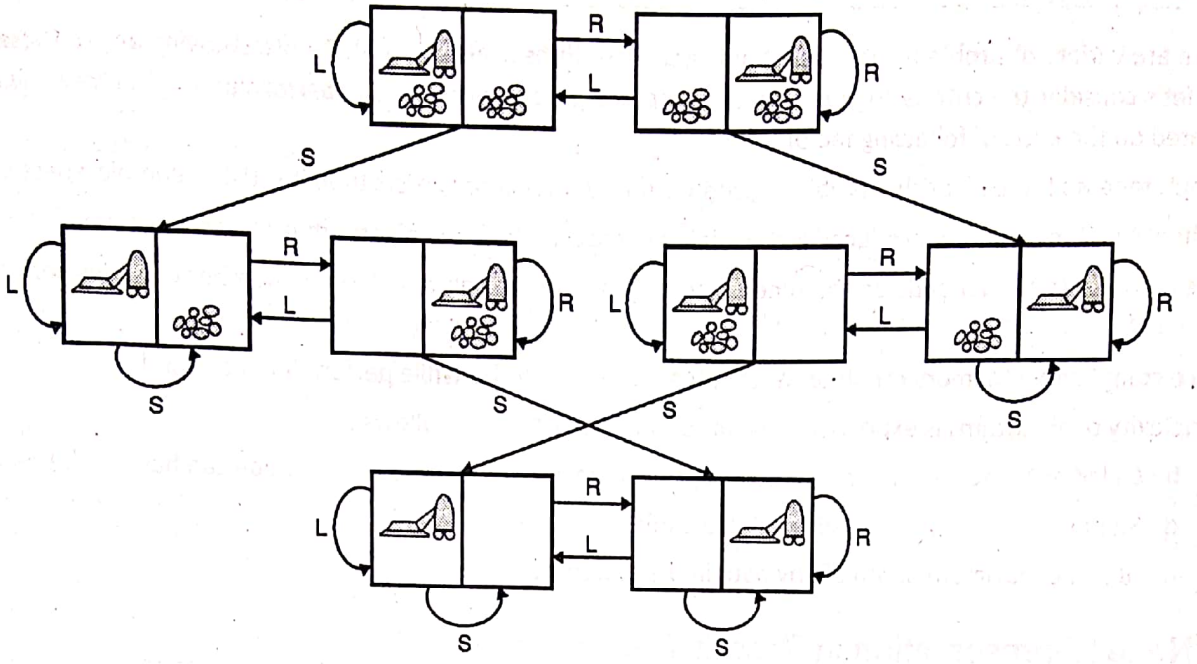


Fig. 2.2.2 : The state space for vacuum world

4. **Goal Test :** The aim of the vacuum cleaner is to clean both the blocks. Hence the goal test if [A, Clean] and [B, Clean].

5. **Path Cost :** Assuming that each action/ step costs 1 unit cost. The path cost is number of actions/ steps taken.

4. Example of real time problems

- There are varieties of real time problems that can be formulated and solved by searching. Robot Navigation, Rout Finding Problem, Traveling Salesman Problem (TSP), VLSI design problem, Automatic Assembly Sequencing, etc. are few to name.
- There are number of applications for route finding algorithms. Web sites, car navigation systems that provide driving directions, routing video streams in computer networks, military operations planning, and airline travel-planning systems are few to name. All these systems involve detailed and complex specifications.
- For now, let us consider a problem to be solved by a travel planning web site; the airline travel problem.
- **State :** State is represented by airport location and current date and time. In order to calculate the path cost state may also record more information about previous segments of flights, their fare bases and their status as domestic or international.

1. **Initial state :** This is specified by the user's query, stating initial location, date and time.

2. **Actions :** Take any flight from the current location, select seat and class, leaving after the current time, leaving enough time for within airport transfer if needed.

3. **Successor function :** After taking the action i.e. selecting fight, location, date, time; what is the next location date and time reached is denoted by the successor function. The location reached is considered as the current location and the flight's arrival time as the current time.

4. **Goal test :** Is the current location the destination location?

5. **Path cost :** In this case path cost is a function of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards and so on.

2.3 Measuring Performance of Problem Solving Algorithm / Agent

There are variety of problem solving methods and algorithms available in AI. Before studying any of these algorithms in detail, let's consider the criteria to judge the efficiency of those algorithms. The performance of all these algorithms can be evaluated on the basis of following factors.

1. **Completeness :** If the algorithm is able to produce the solution if one exists then it satisfies completeness criteria.
2. **Optimality :** If the solution produced is the minimum cost solution, the algorithm is said to be optimal.
3. **Time complexity :** It depends on the time taken to generate the solution. It is the number of nodes generated during the search.
4. **Space complexity :** Memory required to store the generated nodes while performing the search.

Complexity of algorithms is expressed in terms of three quantities as follows :

1. **b :** Called as branching factor representing maximum number of successors a node can have in the search tree.
2. **d :** Stands for depth of the shallowest goal node.
3. **m :** It is the maximum depth of any path in the search tree.

2.4 Node Representation in Search Tree

- In order to carry out search, first we need to build the search tree. The nodes are the various possible states in the state space.
- The connectors are the indicators of which all states are directly reachable from current state, based on the successor function.
- Thus the parent child relation is build and the search tree can be generated. Fig. 2.4.1 shows the representation of a tree node as a data structure in 8-puzzle problem.

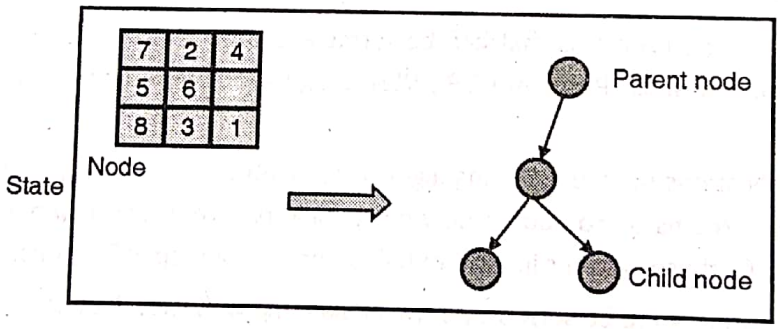


Fig. 2.4.1 : Node representation of state in searching

- Node is the data structure from which the search tree is constructed. Each node has a parent, a state, and children nodes directly reachable from that node.
- For each node of the tree, we can have following structure components :
  1. **State / Value :** The state in the state space to which the node corresponds or value assigned to the node.
  2. **Parent node :** The node in the search tree that generated this node.
  3. **Number of children :** Indicating number of actions that can be taken to generate next states (children nodes).
  4. **Path cost :** The cost of the path from the initial state to the node.

2.5 Uninformed Search

Q. Write short note on Uniform search. MU - Dec. 14

(MU - Dec. 14, 2.5 Marks)

- Why is it called uninformed search? What is not been informed about the search?

- The term "uninformed" means they have only information about what is the start state and the end state along with the problem definition.
- These techniques can generate successor states and can distinguish a goal state from a non-goal state.
- All these search techniques are distinguished by the order in which nodes are expanded.
- The uninformed search techniques also called as "blind search".

2.6 Depth First Search (DFS)

MU - Dec. 12

Q. Explain Depth-first search using suitable example. **(Dec. 12, 4 Marks)**

2.6.1 Concept

- In depth-first search, the search tree is expanded depth wise; i.e. the deepest node in the current branch of the search tree as expanded. As the leaf node is reached, the search backtracks to previous node. The progress of the search is illustrated in Fig.2.6.1.
- The explored nodes are shown in light gray. Explored nodes with no descendants in the fringe are removed from memory. Nodes at depth three have no successors and M is the only goal node.

Process

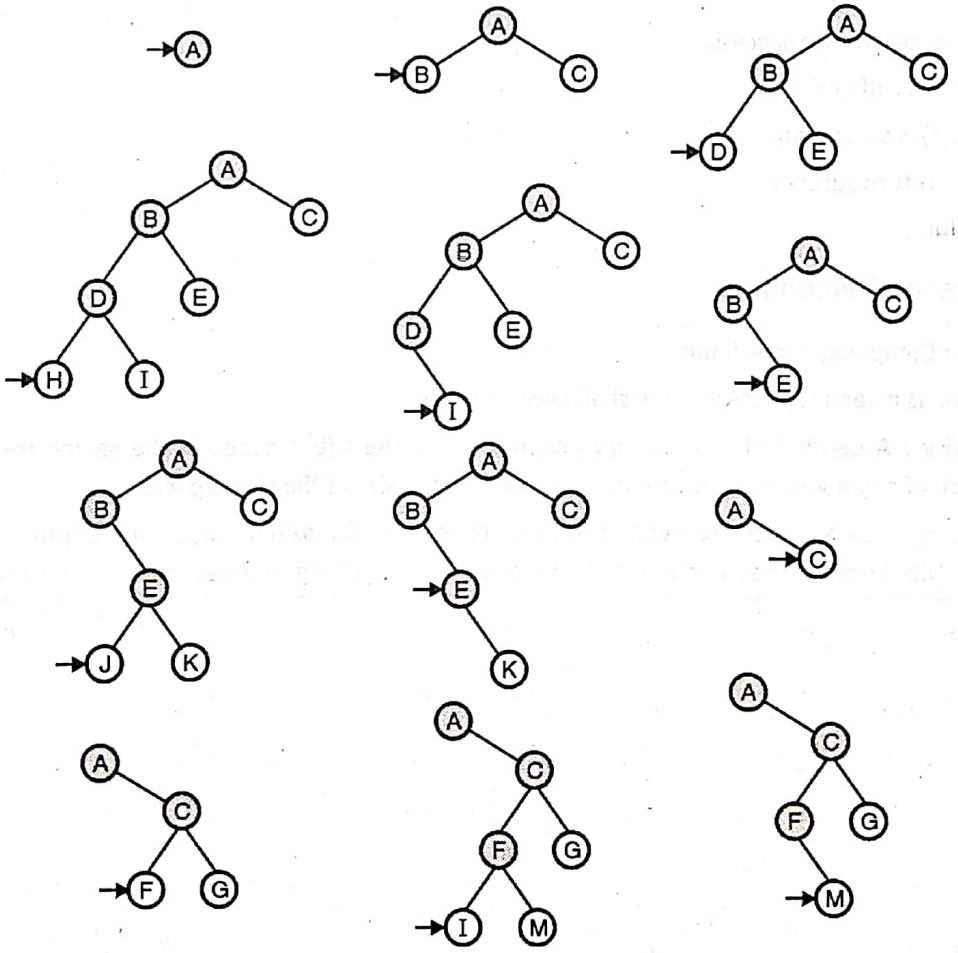


Fig. 2.6.1 : Working of Depth first search on a binary tree

2.6.2 Implementation

DFS uses a LIFO fringe i.e. stack. The most recently generated node, which is on the top in the fringe, is chosen first for expansion. As the node is expanded, it is dropped from the fringe and its successors are added. So when there are no more successors to add to the fringe, the search "back tracks" to the next deepest node that is still unexplored.DFS can be implemented in two ways, recursive and non-recursive. Following is the algorithm for the same.

2.6.3 Algorithm

(a) Non recursive implementation of DFS

1. Push the root node on a stack
2. while (stack is not empty)
   (a) Pop a node from the stack;
      (i) if node is a goal node then return success;
      (ii) push all children of node onto the stack;
3. return failure

(b) Recursive implementation of DFS

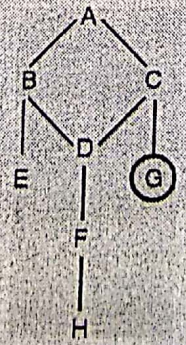
DFS(c) :

1. If node is a goal, return success;
2. for each child c of node
   (a) if DFS(c) is successful,
      (i) return success
3. return failure;

2.6.4 Performance Evaluation

- **Completeness :** Complete, if m is finite.
- **Optimality :** No, as it cannot guarantee the shallowest solution.
- **Time Complexity :** A depth first search, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space.
- **Space Complexity :** For a search tree with branching factor b and maximum depth m, depth first search requires storage of only $O(b^m)$ nodes, as at a time only the branch, which is getting explored, will reside in memory.

Ex. 2.6.1 : Consider following graph.



Starting from state A execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties. **MU - May 16. 10 Marks**

Soln. :

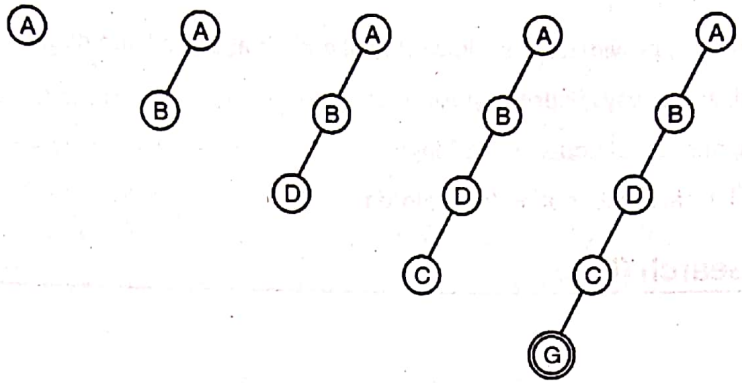


Fig. P. 2.6.1

2.7 Breadth First Search (BFS)

MU - May 14

Q. Explain breadth first algorithm. **(May 14, 10 Marks)**

2.7.1 Concept

- As the name suggests, in breadth-first search technique, the tree is expanded breadth wise.
- The root node is expanded first, then all the successors of the root node are expanded, then their successors, and so on.
- In turn, all the nodes at a particular depth in the search tree are expanded first and then the search will proceed for the next level node expansion.
- Thus, the shallowest unexpanded node will be chosen for expansion. The search process of BFS is illustrated in Fig. 2.7.1.

2.7.2 Process

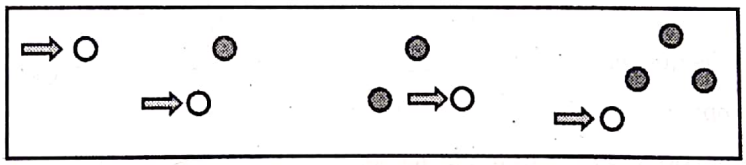


Fig. 2.7.1 : Working of BFS on binary tree

2.7.3 Implementation

- In BFS we use a FIFO queue for the fringe. Because of which the newly inserted nodes in the fringe will automatically be placed after their parents.
- Thus, the children nodes, which are deeper than their parents, go to the back of the queue, and old nodes, which are shallower, get expanded first. Following is the algorithm for the same.

2.7.4 Algorithm

1. Put the root node on a queue
2. while (queue is not empty)
   (a) remove a node from the queue
      (i) if (node is a goal node) return success;
      (ii) put all children of node onto the queue;
3. return failure;

2.7.5 Performance Evaluation

- **Completeness :** It is complete, provided the shallowest goal node is at some finite depth.
- **Optimality :** It is optimal, as it always finds the shallowest solution.
- **Time complexity :** $O(b^d)$, number of nodes in the fringe.
- **Space complexity :** $O(b^d)$, total number of nodes explored.

2.8 Uniform Cost Search (UCS)

2.8.1 Concept

- Uniform cost search is a breadth first search with all paths having same cost. To make it work in real time conditions we can have a simple extension to the basic implementation of BFS. This results in an algorithm that is optimal with any path cost.
- In BFS as we always expand the shallowest node first; but in uniform cost search, instead of expanding the shallowest node, the node with the lowest path cost will be expanded first. The implementation details are as follow.

2.8.2 Implementation

- Uniform cost search can be achieved by implementing the fringe as a priority queue ordered by path cost. The algorithm shown below is almost same as BFS; except for the use of a priority queue and the addition of an extra check in case a shorter path to any node is discovered.
- The algorithm takes care of nodes which are inserted in the fringe for exploration, by using a data structure having priority queue and hash table.
- The priority queue used here contains total cost from root to the node. Uniform cost search gives the minimum path cost the maximum priority. The algorithm using this priority queue is the following.

2.8.3 Algorithm

- Insert the root node into the queue.
- While the queue is not empty :
  - (i) Dequeue the maximum priority node from the queue.
    (If priorities are same, alphabetically smaller node is chosen)
  - (ii) If the node is the goal node, print the path and exit.
    Else
- Insert all the children of the dequeued node, with their total costs as priority.
- The algorithm returns the best cost path which is encountered first and will never go for other possible paths. The solution path is optimal in terms of cost.
- As the priority queue is maintained on the basis of the total path cost of node, the algorithm never expands a node which has a cost greater than the cost of the shortest path in the tree.
- The nodes in the priority queue have almost the same costs at a given time, and thus the name "Uniform Cost Search".

2.8.4 Performance Evaluation

- **Completeness :** Completeness is guaranteed provided the cost of every step exceeds some small positive constant.
- **Optimality :** It produces optimal solution as nodes are expanded in order of their path cost.
- **Time complexity :** Uniform-cost search considers path costs rather than depths; so its complexity is does not merely depends on b and d. Hence we consider C* be the cost of the optimal solution, and assume that every action costs at least €. Then the algorithm's worst-case time and space complexity is $O(b^{C^*/\epsilon})$, which can be much greater than bd.
- **Space complexity :** $O(b^{C^*/\epsilon})$, indicating number of node in memory at execution time.

2.9 Depth Limited Search (DLS)

2.9.1 Concept

- In order to avoid the infinite loop condition arising in DFS, in depth limited search technique, depth-first search is carried out with a predetermined depth limit.
- The nodes with the specified depth limit are treated as if they don't have any successors. The depth limit solves the infinite-path problem.
- But as the search is carried out only till certain depth in the search tree, it introduces problem of incompleteness.
- Depth-first search can be viewed as a special case of depth-limited search with depth limit equal to the depth of the tree. The process of DLS is depicted in Fig. 2.9.1.

2.9.2 Process

If depth limit is fixed to 2, DLS carries out depth first search till second level in the search tree.

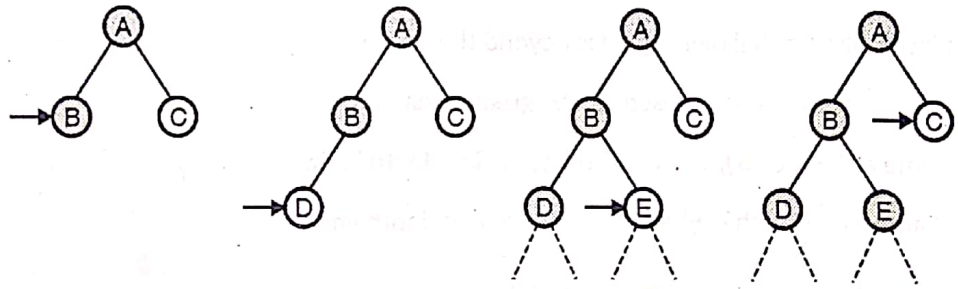


Fig. 2.9.1 : DIS working with depth limit

2.9.3 Implementation

- As in case of DFS in DLS we can use the same fringe implemented as queue.
- Additionally the level of each node needs to be calculated to check whether it is within the specified depth limit. Depth-limited search can terminate with two conditions :
  1. If the solution is found.
  2. If there is no solution within given depth limit.

2.9.4 Algorithm

- Determine the start node and the search depth.
- Check if the current node is the goal node

If not : Do nothing

If yes : return

- Check if the current node is within the specified search depth

If not : Do nothing

If yes : Expand the node and save all of its successors in a stack.

- Call DLS recursively for all nodes of the stack and go back to Step 2.

2.9.5 Pseudo Code

```
booleanDLS(Node node, int limit, intdepth)
{
    if (depth > limit) return failure;
    if (node is a goal node) return success;
    for each child of node
    {
        if (DLS(child, limit, depth + 1))
        return success;
    }
    return failure;
}
```

2.9.6 Performance Evaluation

- **Completeness :** Its incomplete if shallowest goal is beyond the depth limit.
- **Optimality :** Non optimal, as the depth chosen can be greater than d.
- **Time complexity :** Same as DFS, O (b^l), where l is the specified depth limit.
- **Space complexity :** Same as DFS, O(b^l), where l is the specified depth limit.

2.10 Depth First Iterative Deepening (DFID)

2.10.1 Concept

- Iterative deepening depth first search is a combination of BFS and DFS. In DFID search happens depth wise but, at a time the depth limit will be incremented by one. Hence iteratively it deepens down in the search tree.
- It eventually turns out to be the breadth-first search as it explores a complete layer of new nodes at each iteration before going on to the next layer.
- It does this by gradually increasing the depth limit-first 0, then 1, then 2, and so on-until a goal is found; and thus guarantees the optimal solution. Iterative deepening combines the benefits of depth-first and breadth-first search.
- The search process is depicted in Fig. 2.10.1.

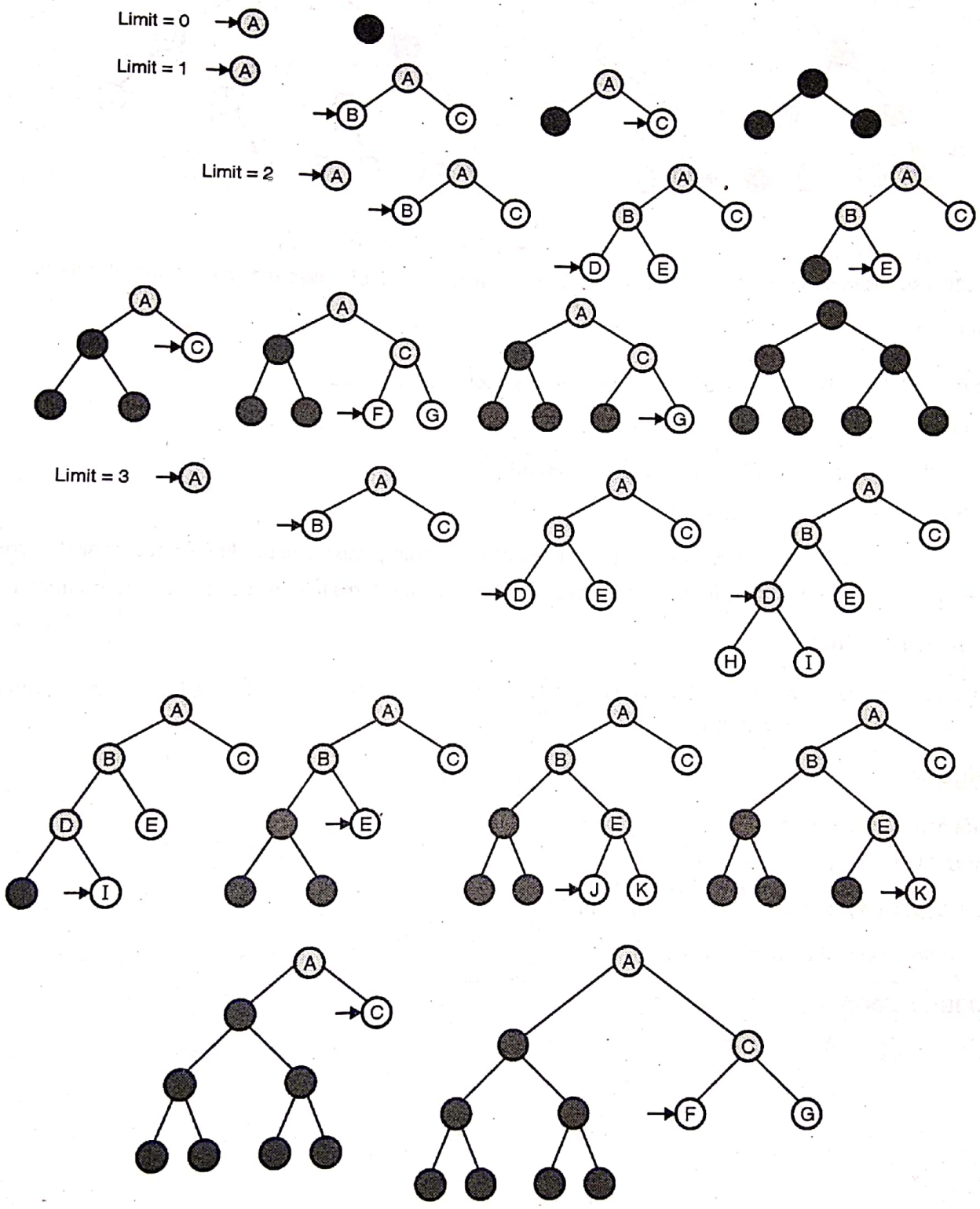


Fig.2.10.1 : Search process in DFID (contd...)

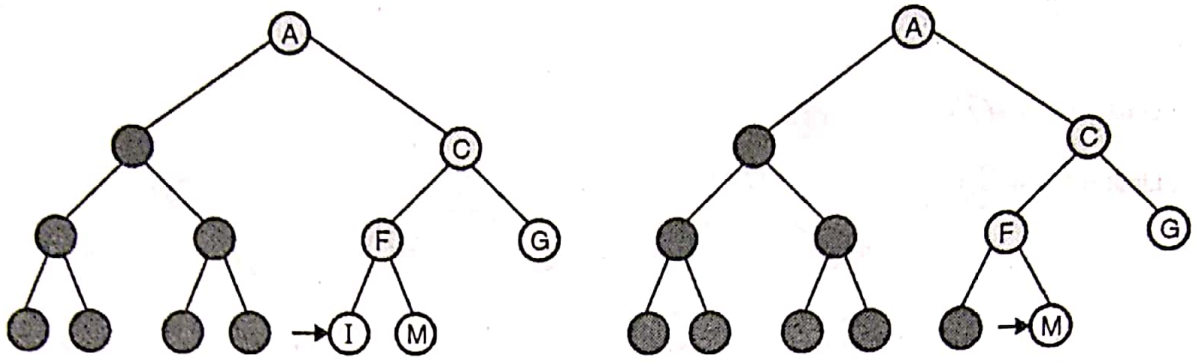


Fig. 2.10.1 : Search process in DFID

– Fig. 2.10.1 shows four iterations of on a binary search tree, where the solution is found on the fourth iteration.

2.10.2 Process

function 1iterative : Depending search (problem) **returns** a solution, or failure

for depth = 0 to ∞ do

result ← Depth – Limited – Search (problem, depth)

if result ≠ cutoff **then return** result

Fig. 2.10.1 the iterative depending search algorithm, which repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.

2.10.3 Implementation

It has exactly the same implementation as that of DLS. Additionally, iterations are required to increment the depth limit by one in every recursive call of DLS.

2.10.4 Algorithm

– Initialize depth limit to zero.

– Repeat Until the goal node is found.

(a) Call Depth limited search with new depth limit.

(b) Increment depth limit to next level.

2.10.5 Pseudo Code

```
DFID()
{
    limit = 0;
    found = false;
    while (not found)
    {
       found = DLS(root, limit, 0);
       limit = limit + 1;
    }
}
```

2.10.6 Performance Evaluation

- **Completeness :** DFID is complete when the branching factor b is finite.
- **Optimality :** It is optimal when the path cost is a non-decreasing function of the depth of the node.
- **Time complexity :**
  - Do you think in DFID there is a lot of wastage of time and memory in regenerating the same set of nodes again and again ?
  - It may appear to be waste of memory and time, but it's not so. The reason is that, in a search tree with almost same branching factor at each level, most of the nodes are in the bottom level which are explores very few times as compared to those on upper level.
  - The nodes on the bottom level that is level 'd' are generated only once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. Hence the time complexity is $O(b^d)$.
- **Space complexity :** Memory requirements of DFID are modest, i.e. $O(b^d)$.

Note : As the performance evaluation is quite satisfactory on all the four parameters, DFID is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

2.11 Bidirectional Search

2.11.1 Concept

In bidirectional search, two simultaneous searches are run. One search starts from the initial state, called forward search and the other starts from the goal state, called backward search. The search process terminates when the searches meet at a common node of the search tree. Fig. 2.11.1 shows the general search process in bidirectional search.

2.11.2 Process

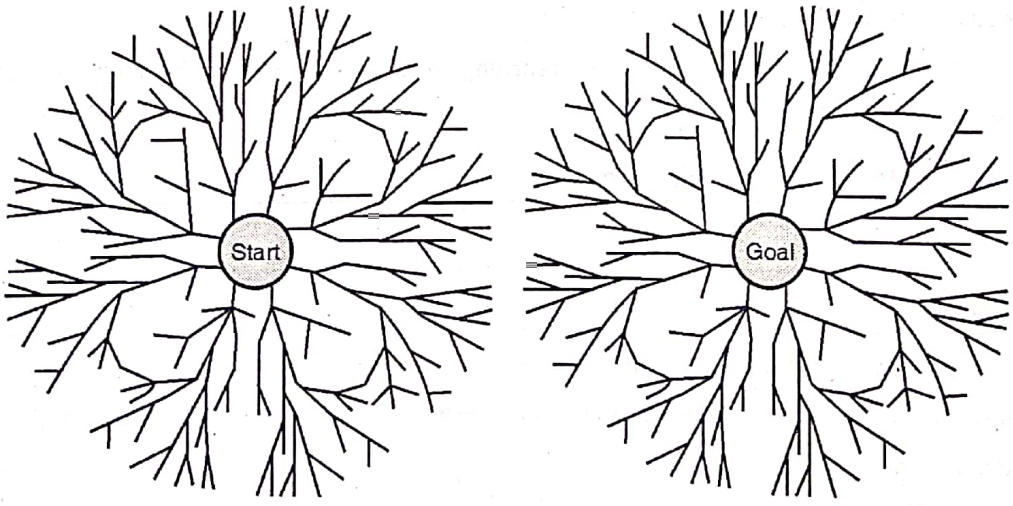


Fig. 2.11.1 : Search process in bidirectional search

2.11.3 Implementation

- In Bidirectional search instead of checking for goal node, one need to check whether the fringes of the two searches intersect; as they do, a solution has been found.

- When each node is generated or selected for expansion, the check can be done. It can be implemented with a hash table, to guarantee constant time.
- For example, consider a problem which has solution at depth d= 6. If we run breadth first search in each direction, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. If b= 10.
- This requires a total of 2,220 node generations, as compared with 1,111,110 for a standard breadth-first search.

2.11.4 Performance Evaluation

- **Completeness :** Yes, if branching factor b is finite and both directions use breadth first search.
- **Optimality :** Yes, if all costs are identical and both directions use breadth first search.
- **Time complexity :** Time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$.
- **Space complexity** : As at least one of the two fringes need to kept in memory to check for the common node, the space complexity is $O(b^{d/2})$.

2.11.5 Pros of Bidirectional Search

- It is much more efficient.
- Reduces space and time requirements as, we perform two $b^{d/2}$ searches, instead of one b^d search.
- **Example :**
  - Suppose b = 10, d = 6. Breadth first search will examine10^6 = 1, 000, 000 nodes.
  - Bidirectional search will examine2×10^3 = 2, 000 nodes.
- One can combine different search strategies in different directions to avail better performance.

2.11.6 Cons of Bidirectional Search

- The search requires generating predecessors of states.
- Overhead of checking whether each new node appears in the other search is involved.
- For large d, is still impractical!
- For two bi-directional breadth-first searches, with branching factor b and depth of the solution d we have memory requirement of $b^{d/2}$ for each search.

2.12 Comparing Different Techniques

Q. Compare different uniformed search strategies. **MU - May 13, Dec. 14**
(May 13, Dec. 14, 10 Marks)

Table 2.12.1 depicts the comparison of all uninformed search techniques basis on their performance evaluation. As we know, the algorithms are evaluated on four criteria viz. completeness, optimality, time complexity and space complexity. The notations used are as follows :

- b : Branching factor
- d : Depth of the shallowest solution
- m : Maximum depth of the search tree
- l : Depth limit

Table 2.12.1 : Comparison of tree-search strategies basis on performance Evaluation

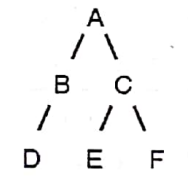
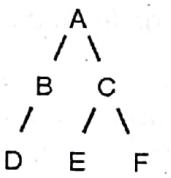
Parameters	BFS	Uniform Cost	DFS	DLS	DFID	Bidirectional
Completeness	Yes	Yes	No	No	Yes	Yes
Optimality	Yes	Yes	No	No	Yes	Yes
Time Complexity	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$

2.12.1 Difference between Unidirectional and Bidirectional Search

Sr. No.	Unidirectional search method	Bidirectional search method
1.	These methods use search tree, start node and goal node as input for starting search.	These methods have additional information about the search tree nodes, along with the start and goal node.
2.	They use only the information from the problem definition.	They incorporate additional measure of a potential of a specific state to reach the goal.
3.	Sometimes these methods use past explorations, e.g. cost of the path generated so far.	All these methods use a potential of a state (node) to reach a goal is measured through heuristic function.
4.	All unidirectional techniques are based on the pattern of exploration of nodes in the search tree.	All bidirectional search techniques totally depend on the evaluated value of each node generated by heuristic function.
5.	In real time problems uninformed search techniques can be costly with respect to time and space.	In real time problems informed search techniques are cost effective with respect to time and space.
6.	Comparatively more number of nodes will be explored in these methods.	As compared to uninformed techniques less number of nodes are explored in this case.
7.	**Example :** Breadth First Search Depth First search Uniform Cost search, Depth Limited search, Iterative Deepening DFS	**Example :** Hill Climbing search Best First search, A* Search, IDA* search, SMA* search

2.12.2 Difference between BFS and DFS

Sr. No.	BFS	DFS
1.	BFS Stands for "Breadth First Search".	DFS stands for "Depth First Search".
2.	BFS traverses the tree level wise.i.e. each node near to root will be visited first. The nodes are explored left to right.	DFS traverses tree depth wise. i.e. nodes in particular branch are visited till the leaf node and then search continues branch by branch from left to right in the tree.

Sr. No.	BFS	DFS
3.	Breadth First Search is implemented using queue which is FIFO list.	Depth First Search is implemented using Stack which is LIFO list.
4.	This is a single step algorithm, wherein the visited vertices are removed from the queue and then displayed at once.	This is two step algorithm. In first stage, the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped out.
5.	BFS requires more memory compare to DFS.	DFS require less memory compare to BFS.
6.	Applications of BFS : To find Shortest path Single Source & All pairs shortest paths In Spanning tree In Connectivity	Applications of DFS : Useful in Cycle detection In Connectivity testing Finding a path between V and W in the graph. Useful in finding spanning trees and forest.
7.	BFS always provides the shallowest path solution.	DFS does not guarantee the shallowest path solution.
8.	No backtracking is required in BFS.	Backtracking is implemented in DFS.
9.	BFS is optimal and complete if branching factor is finite.	DFS is neither complete nor optimal even in case of finite branching factor.
10.	BFS can never get trapped into infinite loops.	DFS generally gets trapped into infinite loops, as search trees are dense.
11.	Example :  A, B, C, D, E, F	Example :  A, B, D, C, E, F

2.13 Informed Search Techniques

MU - Dec. 14

Q. Write short note on Inform search. **(Dec. 14, 2.5 Marks)**

- Informed searching techniques is a further extension of basic un-informed search techniques. The main idea is to generate additional information about the search state space using the knowledge of problem domain, so that the search becomes more intelligent and efficient. The evaluation function is developed for each state, which quantities the desirability of expanding that state in order to reach the goal.
- All the strategies use this evaluation function in order to select the next state under consideration, hence the name "Informed Search". These techniques are very much efficient with respect to time and space requirements as compared to uninformed search techniques.

2.14 Heuristic Function

MU - Dec. 12, Dec. 13, Dec. 14, May 15, Dec. 15

Q. Explain heuristic function with example. **(Dec. 12, Dec. 14, May 15, 5 Marks)**

Q. What is heuristics function ? How will you find suitable heuristic function ? Give suitable example. **(Dec. 13, 10 Marks)**

Q. Define heuristic function. **(Dec. 15, 2 Marks)**

- A heuristic function is an evaluation function, to which the search state is given as input and it generates the tangible representation of the state as output.
- It maps the problem state description to measures of desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.
- It evaluates individual problem state and determines how much promising the state is. Heuristic functions are the most common way of imparting additional knowledge of the problem states to the search algorithm. Fig. 2.14.1 shows the general representation of heuristic function.

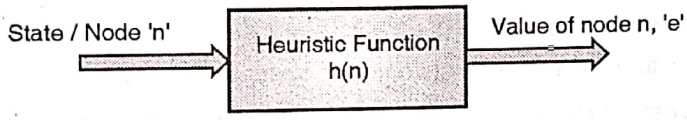


Fig. 2.14.1 : General representation of Heuristic function

- The representation may be the approximate cost of the path from the goal node or number of hopes required to reach to the goal node, etc.
- The heuristic function that we are considering in this syllabus, for a node n is, h(n) = estimated cost of the cheapest path from the state at node n to a goal state.
- **Example :** For the Travelling Salesman Problem, the sum of the distances traveled so far can be a simple heuristic function.
- Heuristic function can be of two types depending on the problem domain. It can be a **Maximization Function** or **Minimization function** of the path cost.
- In maximization types of heuristic, greater the cost of the node, better is the node while; in case of minimization heuristic, lower is the cost, better is the node. There are heuristics of every general applicability as well as domain specific. The search strategies are general purpose heuristics.
- It is believed that in general, a heuristic will always lead to faster and better solution, even though there is no guarantee that it will never lead in the wrong direction in the search tree.
- Design of heuristic plays a vital role in performance of search.
- As the purpose of a heuristic function is to guide the search process in the most profitable path among all that are available; a well designed heuristic functions can provides a fairly good estimate of whether a path is good or bad.
- However in many problems, the cost of computing the value of a heuristic function would be more than the effort saved in the search process. Hence generally there is a trade-off between the cost of evaluating a heuristic function and the savings in search that the function provides.
- So, are you ready to think of your own heuristic function definitions? Here is the word of caution. See how the function definition impact.
- Following are the examples demonstrate how design of heuristic function completely alters the scenario of searching process.

2.14.1 Example of 8-puzzle Problem

- Remember 8-puzzle problem? Can we estimate the number of steps required to solve an 8-puzzle from a given state?? What about designing a heuristic function for it?

7	5	4
5		6
8	3	1

Start state

	1	2
3	4	5
6	7	8

Goal State

Fig. 2.14.2 : A scenario of 8-puzzle problem

- Two simple heuristic functions are :
  - h_1 = the number of misplaced tiles. This is also known as the Hamming Distance. In the Fig. 2.14.2 example, the start state has $h_1 = 8$. Clearly, h_1 is an acceptable heuristic because any tile that is out of place will have to be moved at least once, quite logical. Isn't it?
  - h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot be moved diagonally, the distance counted is the sum of horizontal and vertical distances. This is also known as the Manhattan Distance. In the Fig. 2.14.2, the start state has $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$. Clearly, h_2 is also an admissible heuristic because any move can, at best, move one tile one step closer to the goal.
- As expected, neither heuristic overestimates the true number of moves required to solve the puzzle, which is 26 ($h_1 + h_2$). Additionally, it is easy to see from the definitions of the heuristic functions that for any given state, h_2 will always be greater than or equal to h_1. Thus, we can say that h_2 dominates h_1.

2.14.2 Example of Block World Problem

MU - Dec. 15

Q. Give an example heuristics function for block world problem. **(Dec. 15, 3 Marks)**

Q. Find the heuristics value for a particular state of the blocks world problem. **(Dec. 15, 5 Marks)**

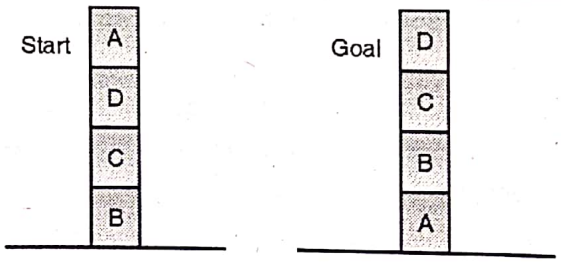


Fig. 2.14.3 : Block problem

- Fig. 2.14.3 depicts a block problem world, where the A, B, C,D letter bricks are piled up on one another and required to be arranged as shown in goal state, by moving one brick at a time. As shown, the goal state with the particular arrangement of blocks need to be attain from the given start state. Now it's time to scratch your head and define a heuristic function that will distinguish start state from goal state. Confused??
- Let's design a function which assigns + 1 for the brick at right position and – 1 for the one which is at wrong position. Consider Fig. 2.14.4

(a) Local heuristic :

- + 1 for each block that is resting on the thing it is supposed to be resting on.
- – 1 for each block that is resting on a wrong thing.

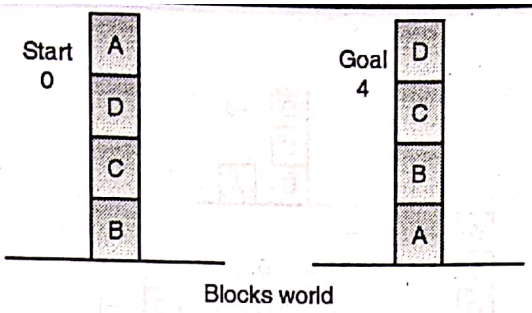


Fig. 2.14.4 : Definition of heuristic function "h_1"

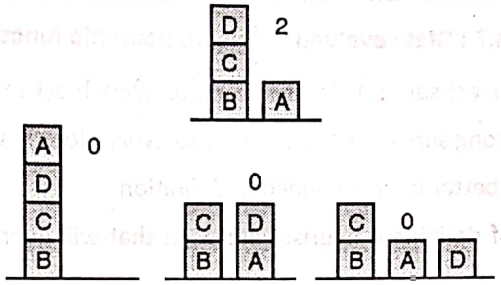


Fig. 2.14.5 : State evaluations using Heuristic function "h_1"

- Fig. 2.14.5 shows the heuristic values generated by heuristic function "h_1" for various different states in the state space. Please observe that, this heuristic is generating same value for different states.
- Due to this kind of heuristic the search may end up in limitless iterations as the state showing most promising heuristic value may not hold true or search may end up in finding an undesirable goal state as the state evaluation may lead to wrong direction in the search tree.
- Let's have another heuristic design for the same problem. Fig. 2.14.6 is depicting a new heuristic function "h_2" definition, in which the correct support structure of each brick is given +1 for each brick in the support structure. And the one not having correct support structure, –1 for each brick in the wrong support structure.

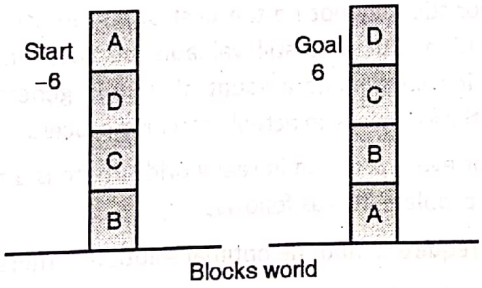


Fig. 2.14.6 : Definition of heuristic function "h_2"

- As we observe in Fig. 2.14.7, the same states are considered again as that of Fig. 2.14.5, but this time using h_2, each one of the state is assigned a unique value generate according to heuristic function h_2.
- Observing this example one can easily understand that, in the second part of the example, search will be carried out smoothly as each unique state is getting a unique value assigned to it.
- This example makes it clear that, the design of heuristic plays a vital role in search process, as the whole search is carried out by considering the heuristic values as basis for selecting the next state to be explored.
- The state having the most promising value to reach to the goal state will be the first prior candidate for exploration, this continues till we find the goal state.

(b) Global heuristic :

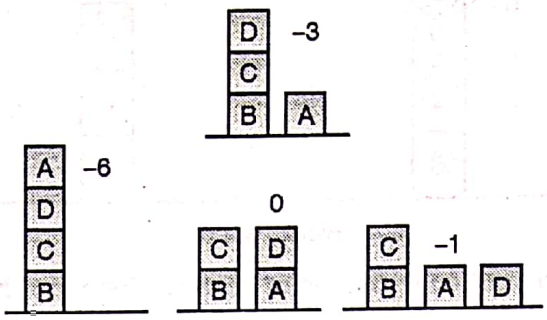


Fig. 2.14.7 : State evaluations using heuristic function "h_2"

- For each block that has the correct support structure : + 1 to every block in the support structure.
- For each block that has the wrong support structure : – 1 to every block in the support structure.
- This leads to a discussion of a better heuristic function definition.
- Is there any particular way of defining a heuristic function that will guarantee a better performance in search process??

2.14.3 Properties of Good Heuristic Function

1. It should generate a unique value for each unique state in search space.
2. The values should be a logical indicator of the profitability of the state in order to reach the goal state.
3. It may not guarantee to find the best solution, but almost always should find a very good solution.
4. It should reduce the search time; specifically for hard problems like travelling salesman problem where the time required is exponential.

- The main objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem, as it's an extra task added to the basic search process.
- The solution produced by using heuristic may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding the solution does not require a prohibitively long time. So we are investing some amount of time in generating heuristic values for each state in search space but reducing the total time involved in actual searching process.
- Do we require to design heuristic for every problem in real world? There is a trade-off criterion for deciding whether to use a heuristic for solving a given problem. It is as follows.
  - **Optimality :** Does the problem require to find the optimal solution, if there exist multiple solutions for the same?
  - **Completeness :** In case of multiple existing solution of a problem, is there a need to find all of them? As many heuristics are only meant to find one solution.
  - **Accuracy and precision :** Can the heuristic guarantee to find the solution within the precision limits? Is the error bar on the solution unreasonably large?
  - **Execution time :** Is it going to affect the time required to find the solution? Some heuristics converge faster than others. Whereas, some are only marginally quicker than classic methods.
- In many AI problems, it is often hard to measure precisely the goodness of a particular solution. But still it is important to keep performance question in mind while designing algorithm. For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is impossible to define this knowledge in such a way that mathematical analysis can be performed.

2.15 Best First Search

2.15.1 Concept

- In depth first search all competing branches are not getting expanded. And breadth first search never gets trapped on dead end paths. If we combine these properties of both DFS and BFS, it would be "follow a single path at a time, but switch paths whenever some competing path look more promising than the current one". This is what the Best First search is..!!
- Best-first search is a search algorithm which explores the search tree by expanding the most promising node chosen according to the heuristic value of nodes. Judea Pearl described best-first search as estimating the promise of node n by a "heuristic evaluation function f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain".
- Efficient selection of the current best candidate for extension is typically implemented using a priority queue. Fig. 2.15.1 depicts the search process of Best first search on an example search tree. The values noted below the nodes are the estimated heuristic values of nodes.

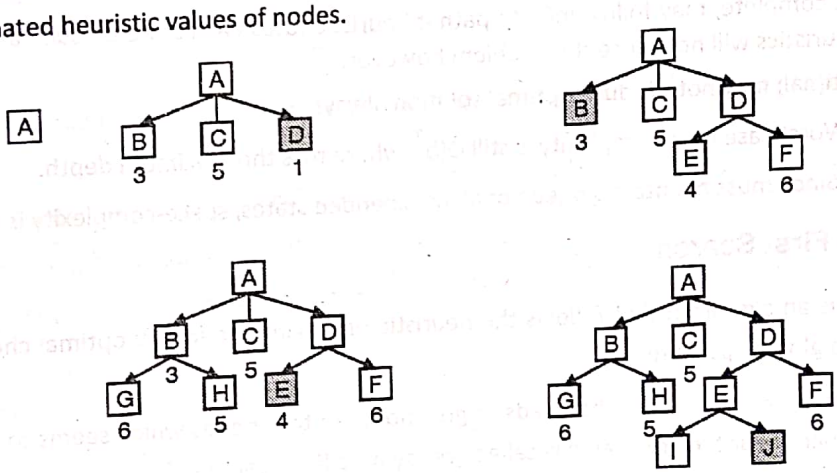


Fig. 2.15.1 : Best first search tree expansion scenario

2.15.2 Implementation

- Best first search uses two lists in order to record the path. These are namely OPEN list and CLOSED list for implementation purpose.
- OPEN list stores nodes that have been generated, but have not examined. This is organized as a priority queue, in which nodes are stored with the increasing order of their heuristic value, assuming we are implementing maximization heuristic. It provides efficient selection of the current best candidate for extension.
- CLOSED list stores nodes that have already been examined. This CLOSED list contains all nodes that have been evaluated and will not be looked at again. Whenever a new node is generated, check whether it has been generated before. If it is already visited before, check its recorded value and change the parent if this new value is better than previous one. This will avoid any node being evaluated twice, and will never get stuck into an infinite loops.

2.15.3 Algorithm : Best First Search

OPEN = [initial state]
CLOSED = []
while OPEN is not empty
do

1. Remove the best node from OPEN, call it n, add it to CLOSED.
2. If n is the goal state, backtrack path to n through recorded parents and return path.
3. Create n's successors.
4. For each successor do :
   a. If it is not in CLOSED and it is not in OPEN : evaluate it, add it to OPEN, and record its parent.
   b. Otherwise, if it is already present in OPEN with different parent node and this new path is better than previous one, change its recorded parent.
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN using this new evaluation.

   done

This algorithm of Best First Search algorithm just terminates when no path is found. An actual implementation would of course require special handling of this case.

2.15.4 Performance Measures for Best first search

- **Completeness :** Not complete, may follow infinite path if heuristic rates each state on such a path as the best option. Most reasonable heuristics will not cause this problem however.
- **Optimality :** Not optimal; may not produce optimal solution always.
- **Time Complexity :** Worst case time complexity is still $O(b^m)$where m is the maximum depth.
- **Space Complexity :** Since must maintain a queue of all unexpanded states, space-complexity is also $O(b^m)$.

2.15.5 Greedy Best First Search

- A greedy algorithm is an algorithm that follows the heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
- When Best First Search uses a heuristic that leads to goal node, so that nodes which seems to be more promising are expanded first. This particular type of search is called greedy best-first search.
- In greedy best first search algorithm, first successor of the parent is expanded. For the successor node, check the following :
  1. If the successor node's heuristic is better than its parent, the successor is set at the front of the queue, with the parent reinserted directly behind it, and the loop restarts.
  2. Else, the successor is inserted into the queue, in a location determined by its heuristic value. The procedure will evaluate the remaining successors, if any of the parent.
- In many cases, greedy best first search may not always produce an optimal solution, but the solution will be locally optimal, as it will be generated in comparatively less amount of time. In mathematical optimization, greedy algorithms solve combinatorial problems.
- For example, consider the traveling salesman problem, which is of a high computational complexity, works well with greedy strategy as follows. Refer to Fig. 2.15.2. The values written on the links are the straight line distances from the nodes. Aim is to visit all the cities A through F with the shortest distance travelled.
- Let us apply a greedy strategy for this problem with a heuristic as, "At each stage visit an unvisited city nearest to the current city". Simple logic... isn't it? This heuristic need not find a best solution, but terminates in a reasonable number of steps by finding an optimal solution which typically requires unreasonably many steps. Let's verify.

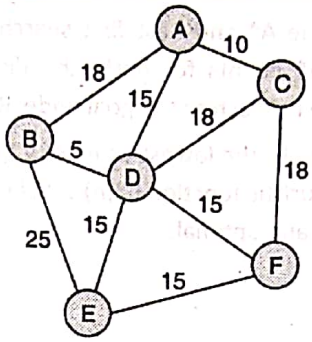


Fig. 2.15.2 : Travelling Salesmen Problem example

- As greedy algorithm, it will always make a local optimal choice. Hence it will select node C first as it found to be the one with less distance from the next non-visited node from node A, and then the path generated will be A→C→D→B→E→F with the total cost = 10 + 18 + 5 + 25 + 15 = 73. While by observing the graph one can find the optimal path and optimal distance the salesman needs to travel. It turns out to be, A→B→D→E→F→C where the cost comes out to be 18 + 5 + 15 + 15 + 18 = 68.

2.15.6 Properties of Greedy Best-First Search

1. **Completeness :** It's not complete as, it can get stuck in loops, also is susceptible to wrong start and quality of heuristic function.
2. **Optimality :** It's not optimal; as it goes on selecting a single path and never checks for other possibilities.
3. **Time Complexity :** $O(b^m)$, but a good heuristic can give dramatic improvement.
4. **Space Complexity :** $O(b^m)$, It needs to keep all nodes in memory.

2.16 A* Search

MU - May 13, Dec. 13, May 14, Dec. 14, May 15

Q.	Explain A* Algorithm. What is the drawback of A* ? Also shows that A* is optimally efficient.	**(May 13, 10 Marks)**
Q.	Describe A* algorithm with merits and demerits.	**(Dec. 13, 10 Marks)**
Q.	Explain A* algorithm with example.	**(May 14, Dec. 14, 10 Marks)**
Q.	Explain A* search with example.	**(May 15, 10 Marks)**

2.16.1 Concept

- A* pronounced as "Aystar" (Hart, 1972) search method is a combination of branch and bound and best first search, combined with the dynamic programming principle.
- It's a variation of Best First search where the evaluation of a state or a node not only depends on the heuristic value of the node but also considers its distance from the start state. It's the most widely known form of best-first search. A* algorithm is also called as OR graph / tree search algorithm.
- In A* search, the value of a node n, represented as f(n) is a combination of g(n), which is the cost of cheapest path to reach to the node from the root node, and h(n), which is the cost of cheapest path to reach from the node to the goal node. Hence f(n) = g(n) + h(n).
- As the heuristic can provide only the estimated cost from the node to the goal we can represent h(n) as h*(n); similarly g*(n) can represent approximation of g(n) which is the distance from the root node observed by A* and the algorithm A* will have,

$$f^*(n) = g^*(n) + h^*(n)$$

- As we observe the difference between the A* and Best first search is that; in Best first search only the heuristic estimation of h(n) is considered while A* counts for both, the distance travelled till a particular node and the estimation of distance need to travel more to reach to the goal node, it always finds the cheapest solution.
- A reasonable thing to try first is the node with the lowest value of g*(n) +h*(n). It turns out that this strategy is more than just reasonable, provided that the heuristic function h*(n) satisfies certain conditions which are discussed further in the chapter. A* search is both complete and optimal.

2.16.2 Implementation

A* does also use both OPEN and CLOSED list.

2.16.3 Algorithm (A*)

1. Initialization OPEN list with initial node; CLOSED= ϕ; g = 0, f = h, **Found = false;**
2. While (OPEN $\neq \phi$ and **Found = false**)

{1

i. Remove the node with the lowest value of **f** from OPEN to CLOSED and call it as a **Best_Node.**

ii If Best_Node = Goal state then **Found = true**

iii. else

{2

a. Generate the **Succof Best_Node**

b. For each **Succ** do

{3

i. Compute g(Succ) = g(Best_Node) + cost of getting from Best_Node to Succ.

ii. If Succ$\in$ OPEN then /* already being generated but not processed */

{4

a. Call the matched node as OLD and add it in the list of Best_Node successors.

b. Ignore the **Succ** node and change the parent of OLD, if required.

- If g(Succ) < g(OLD) then make parent of OLD to beBest_Node and change the values of g and f for OLD
- If g(Succ) >= g(OLD) then ignore

}4

a. If Succ$\in$ CLOSED then /* already processed */

{5

i. Call the matched node as OLD and add it in the list of Best_Node successors.

ii. Ignore the **Succ** node and change the parent of OLD, if required

- If g(Succ) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD.
- Propogate the change to OLD's children using depth first search
- If g(Succ) >= g(OLD) then do nothing

}5

a. If Succ$\notin$ OPEN or CLOSED

{6

i. Add it to the list of Best_Node's successors

ii. Compute f(Succ) = g(Succ) + h(Succ)

iii. Put **Succ** on OPEN list with its f value

$\}^6$

$\}^3$ /* for loop*/

$\}^2$ /* else if */

$\}^1$ /* End while */.

3. If **Found = true** then report the best path else report failure

4. Stop

2.16.4 Behaviour of A* Algorithm

As stated already the success of A* totally depends upon the design of heuristic function and how well it is able to evaluate each node by estimating its distance from the goal node. Let us understand the effect of heuristic function on the execution of the algorithm and how the optimality gets affected by it.

A. Underestimation

- If we can guarantee that heuristic function 'h' never over estimates actual value from current to goal that is, the value generated by h is the always lesser than the actual cost or actual number of hopes required to reach to the goal state. In this case, A* algorithm is guaranteed to find an optimal path to a goal, if one exists.

Example :

f = g + h, Here h is underestimated.

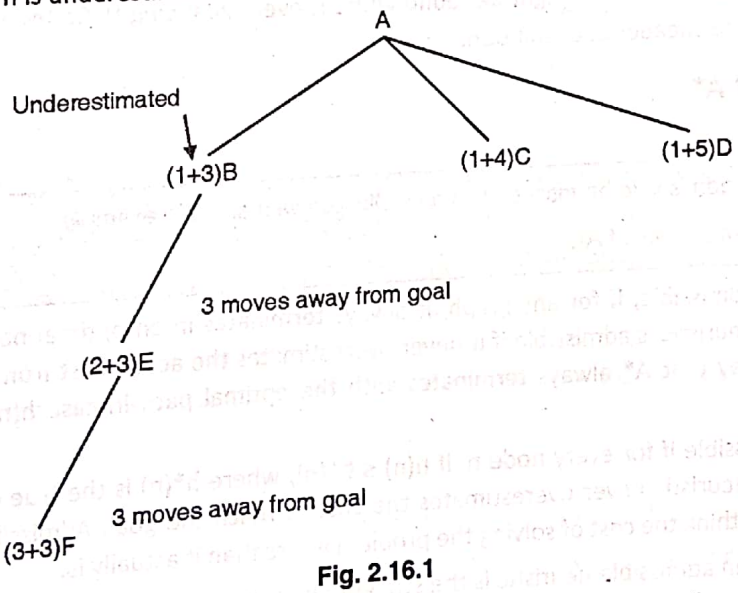


Fig. 2.16.1

- If we consider cost of all arcs to be 1. A is expanded to B, C and D. 'f' values for each node is computed. B is chosen to be expanded to E. We notice that f(E) = f(C) = 5. Suppose we resolve in favor of E, the path currently we are expanding. E is expanded to F. Expansion of a node F is stopped as f(F)=6 so we will now expand node C.
- Hence by underestimating h(B), we have wasted some effort but eventually discovered that B was farther away than we thought. Then we go back and try another path, and will find optimal path.

B. Overestimation

- Here h is overestimated that is, the value generated for each node is greater than the actual number of steps required to reach to the goal node.

Example :

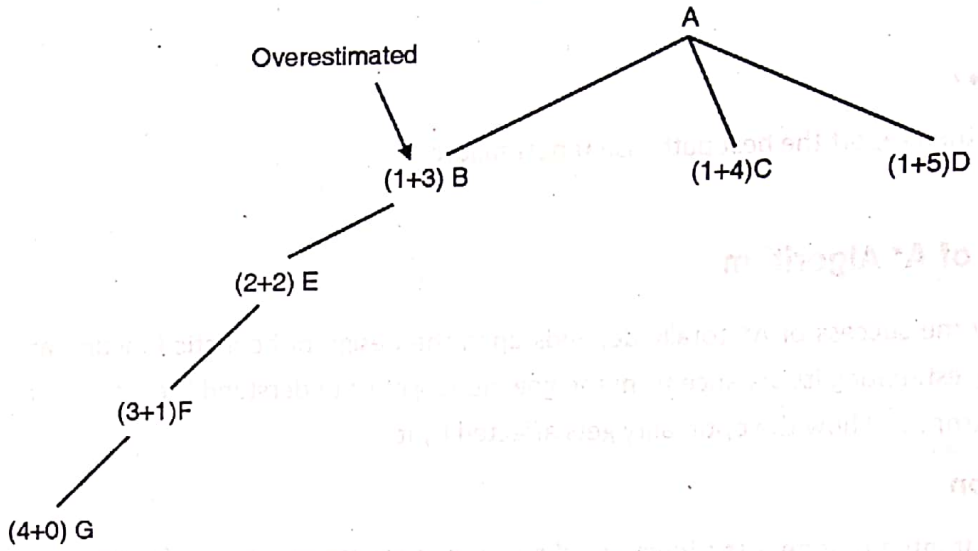


Fig. 2.16.2

- As shown in the example, A is expanded to B, C and D. Now B is expanded to E, E to F and F to G for a solution path of length 4. Consider a scenario when there a direct path from D to G with a solution giving a path of length 2.This path will never be found because of overestimating h(D).
- Thus, some other worse solution might be found without ever expanding D. So by overestimating h, one cannot guarantee to find the cheaper path solution.

2.16.5 Admissibility of A*

MU - Dec. 12, May 14

Q. What do you mean an admissible heuristics function ? Discuss with suitable example. **(Dec. 12, 5 Marks)**

Q. Write short note on admissibility of A*. **(May 14, 5 Marks)**

- A search algorithm is admissible, if for any graph, it always terminates in an optimal path from initial state to goal state, if path exists. A heuristic is admissible if it never overestimates the actual cost from current state to goal state. Alternatively, we can say that A* always terminates with the optimal path in case h(n) is an admissible heuristic function.
- A heuristic h(n) is admissible if for every node n, if $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n. An admissible heuristic never overestimates the cost to reach the goal. Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.
- An obvious example of an admissible heuristic is the straight line distance. Straight line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot overestimate the actual road distance.
  - **Theorem :** If h(n) is admissible, tree search using A* is optimal.
  - **Proof :** Optimality of A*with admissible heuristic.
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

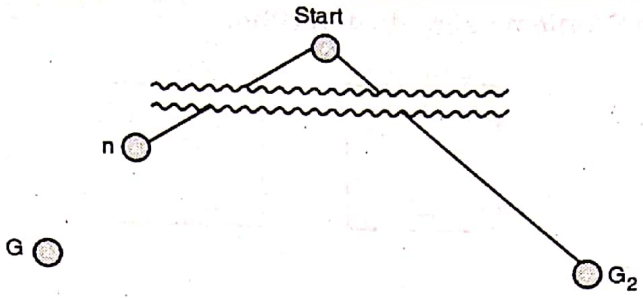


Fig. 2.16.3 : Optimality of A*

$f(G_2)$	= $g(G_2)$	since $h(G_2) = 0$
$g(G_2)$	> $g(G)$	since G_2 is suboptimal
$f(G)$	= $g(G)$	since $h(G) = 0$
$f(G_2)$	> $f(G)$	from above
$h(n)$	≤ $h^*(n)$	since h is admissible
$g(n) + h(n)$	≤ $g(n) + h^*(n)$	
$f(n)$	≤ $f(G)$	

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion.

2.16.6 Monotonicity

MU -May 16

Q. Prove that A* is admissible if it uses Monotone heuristic. (May 16, 5 Marks)

- A heuristic function **h** is monotone or consistent if,
- ∀states X_i and X_j such that X_j is successor of Xi,

$$h(X_i) - h(X_j) \leq \text{cost}(X_i, X_j)$$

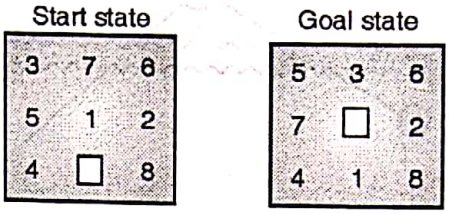
where, cost (X_i, X_j)actual cost of going from X_i to X_j and h (goal) = 0

- In this case, heuristic is locally admissible i.e., consistently finds the minimal path to each state they encounter in the search. The monotone property in other words is that search space which is everywhere locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors. With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter. Each monotonic heuristic is admissible.
- A cost function f(n) is monotone if $f(n) \leq f(\text{succ}(n))$, ∀n.
- For any admissible cost function f, we can construct a monotone admissible function.
- Alternatively, the monotone property : that search space which is everywhere locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors.
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter.

2.16.7 Properties of A*

1. **Completeness :** It is complete, as it will always find solution if one exist.
2. **Optimality :** Yes, it is Optimal.
3. **Time Complexity :** $O(b^m)$, as the number of nodes grows exponentially with solution cost.
4. **Space Complexity :** $O(b^m)$,as it keeps all nodes in memory.

2.16.8 Example : 8 Puzzle Problem using A* Algorithm



Evaluation function - f for EPP

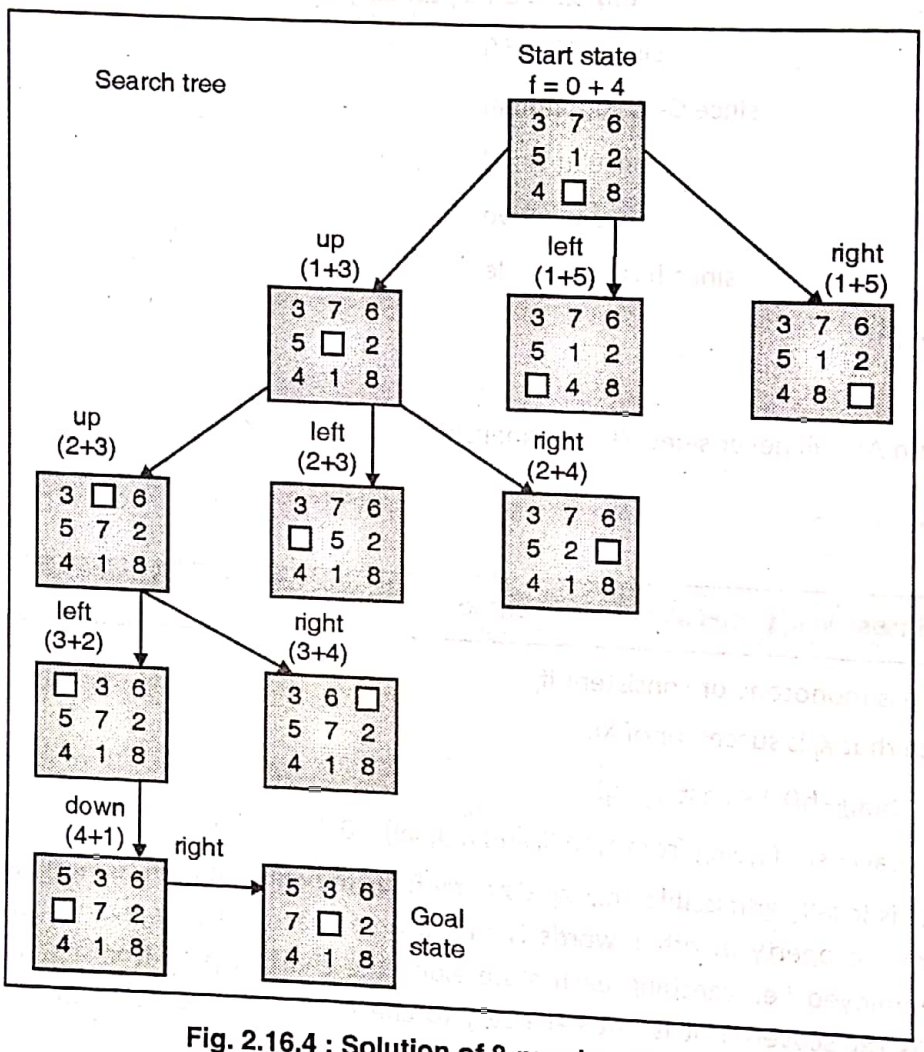


Fig. 2.16.4 : Solution of 8-puzzle using A*

- The choice of evaluation function critically determines search results.
- Consider Evaluation function

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

$h(X)$ = the number of tiles not in their goal position in a given state X

$g(X)$ = depth of node X in the search tree.

- For Initial node f(initial_node) = 4

Ex. 2.16.1 : Consider the graph given in Fig. P.2.16.1 below. Assume that the initial state is S and the goal state is 7. Find a path from the initial state to the goal state using A* Search. Also report the solution cost. The straight line distance heuristic estimates for the nodes are as follows : h(1) = 14, h(2) = 10, h(3) = 8, h(4) = 12, h(5) = 10, h(6) = 10, h(S) = 15.

MU - Dec. 15, 5 Marks

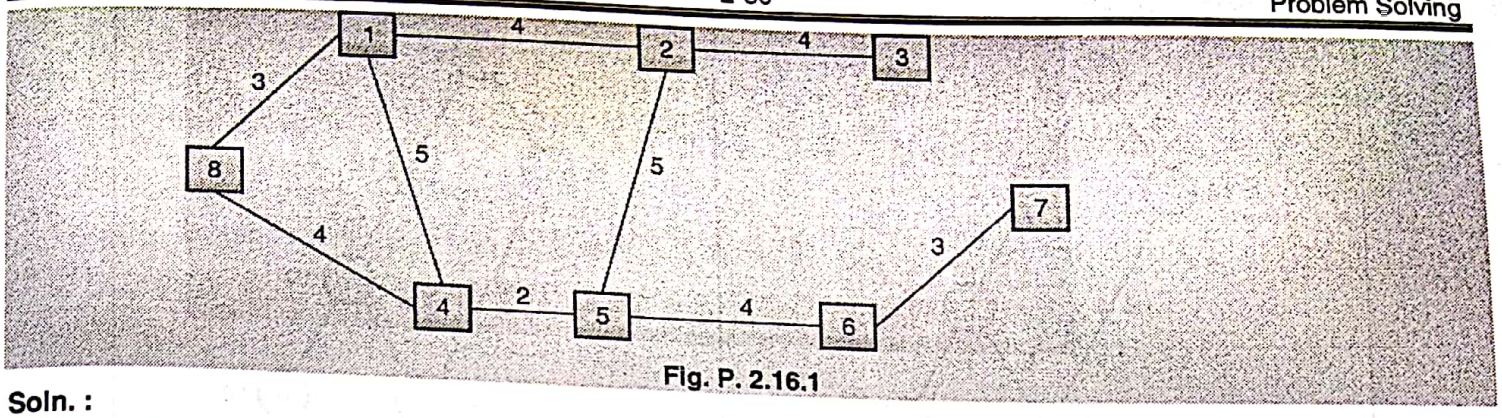
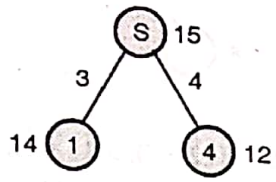


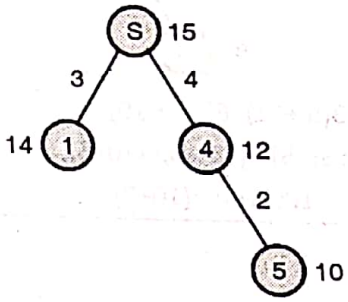
Fig. P. 2.16.1

Soln. :



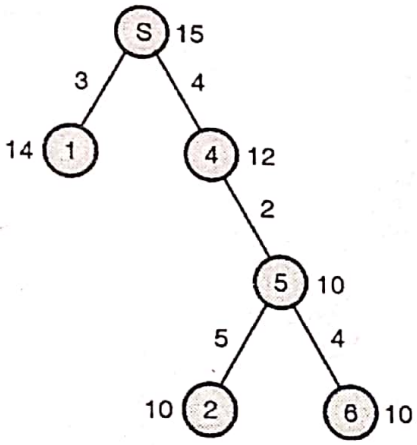
Open : 4(12 + 4), 1(14 + 3)

Closed : S(15)



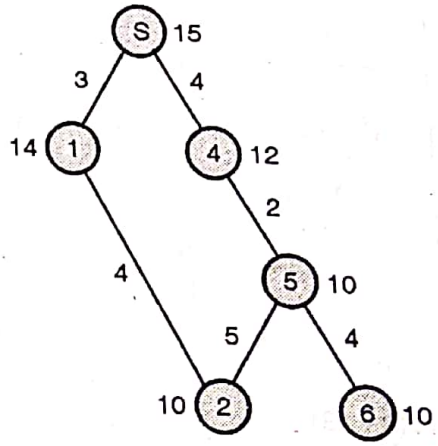
Open : 5(10 + 6), 1(14 + 3)

Closed : S(15), 4 (12 + 4)



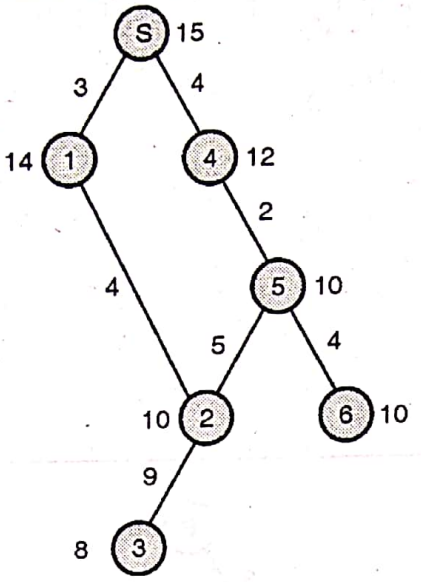
Open : 1(14 + 3) 6(10 + 10) 2(10 + 11)

Closed : S(15) 4(12 + 4) 5(10 + 6)



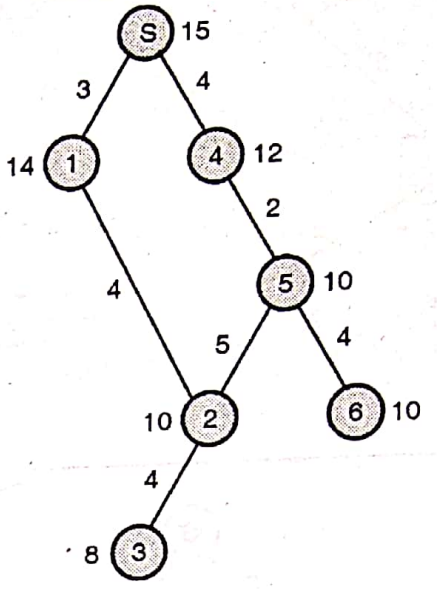
Open : 2(10 + 7) 6(10+10)

Closed : S(5) 4(12 + 4) 5(10 + 6) 1(14 +3)



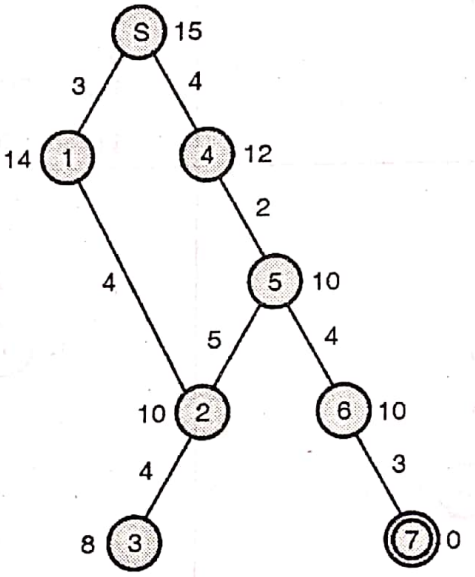
Open : 3(8 +11), 6(10 + 10)

Closed : S(15),4(12+4),5(10+6),
1(14+ 3),2(10+7)



Open : 6(10 + 10)

Closed : s(15), 4(12 + 4), 5(10 + 6),
1(14 + 3) 2(10 + 7) 3(8 + 11)



Open : 7(0 + 13)

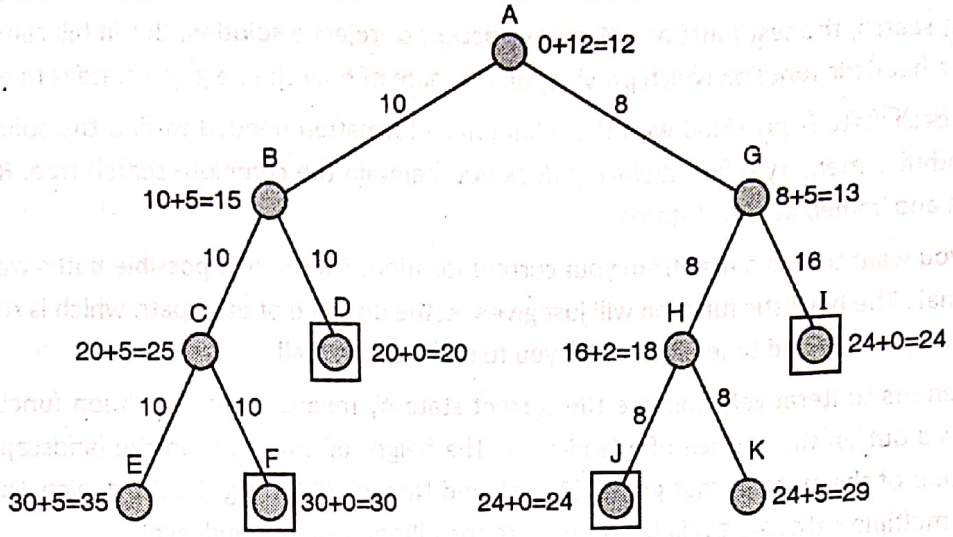
Closed : S(15), 4(12 + 4), 5(10 + 6), 1(14 + 3), 2(10 + 7), 5(8 + 4), 6(10 + 10)

2.16.9 Caparison among Best First Search, A* search and Greedy Best First Search

MU - May 16

Q. Compare following informed searching algorithms based on performance measure with justification : Complete, Optimal, Time complexity and space complexity.
(a)Greedy best first (b) A* (c) recursive best-first (RBFS) **(May 16, 10 Marks)**

Algorithm	Greedy Best First Search	A* search	Best First Search
Completeness	Not complete	complete	Not complete
Optimality	Not optimal	optimal	Not optimal
Time Complexity	$O(b^m)$	$O(b^m)$	O(bm)
Space Complexity	$O(b^m)$	$O(b^m)$	O(bm)



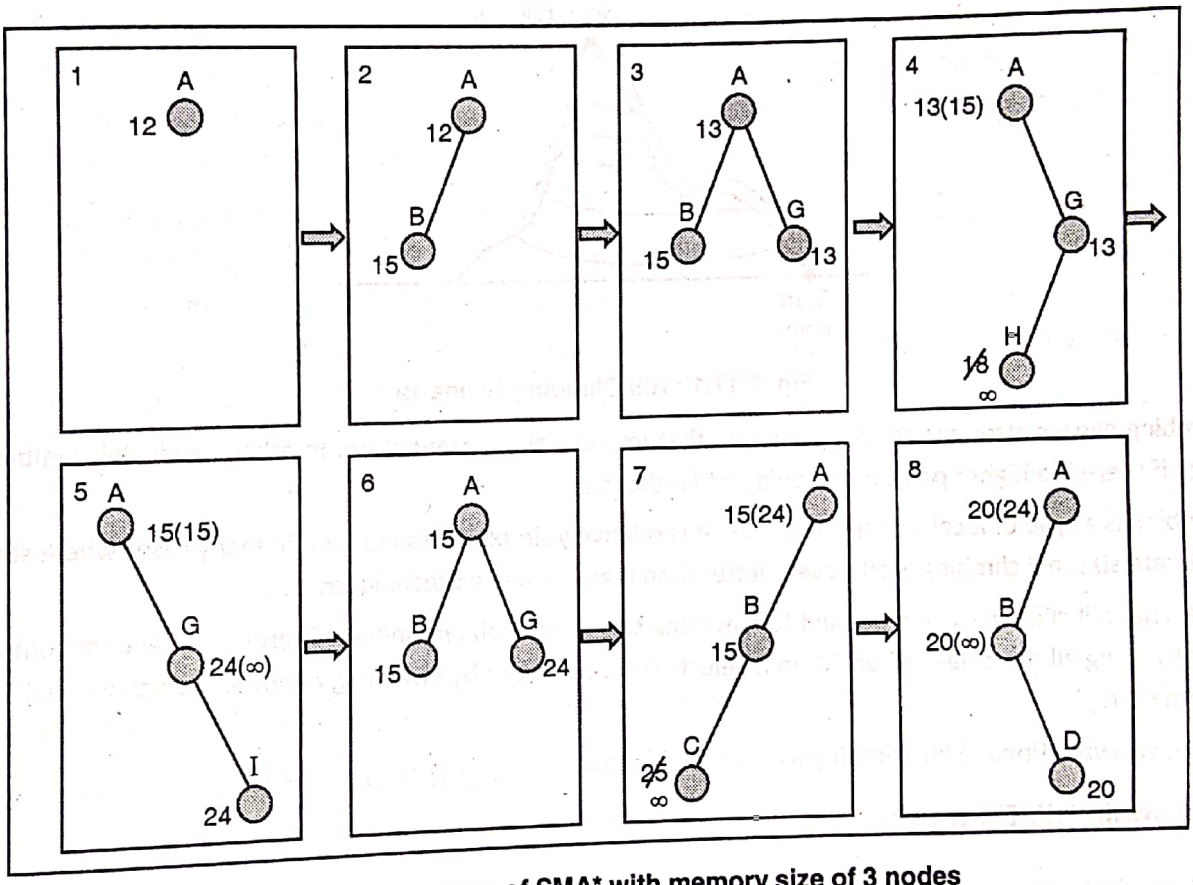


Fig. 2.16.5 : Process of SMA* with memory size of 3 nodes

2.17 Local Search Algorithms and Optimization Problems

2.17.1 Hill Climbing

MU -May 13, Dec. 13,Dec. 14, May 16

Q. Write short note on Hill Climbing algorithm. **(Dec. 13, 5 Marks)**

Q. Describe Hill Climbing algorithm. **(May 13, Dec. 14, 5 Marks)**

Q. Explain Hill- Climbing algorithm with example. **(May 16, 5 Marks)**

- Hill climbing is simply a combination of depth first with generate and test where a feedback is used here to decide on the direction of motion in the search space.
- Hill climbing technique is used widely in artificial intelligence, to solve solving computationally hard problems, which has multiple possible solutions.

- In the depth-first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state.
- In Hill climbing, each state is provided with the additional information needed to find the solution, i.e. the heuristic value. The algorithm is memory efficient since it does not maintain the complete search tree. Rather, it looks only at the current state and immediate level states.
- For example, if you want to find a mall from your current location. There are n possible paths with different directions to reach to the mall. The heuristic function will just give you the distance of each path which is reaching to the mall, so that it becomes very simple and time efficient for you to reach to the mall.
- Hill climbing attempts to iteratively improve the current state by means of an evaluation function. "Consider all the possible states laid out on the surface of a landscape. The height of any point on the landscape corresponds to the evaluation function of the state at that point" (Russell and Norvig, 2003). Fig. 2.17.1 depicts the typical hill climbing scenario, where multiple paths are available to reach to the hill top from ground level.

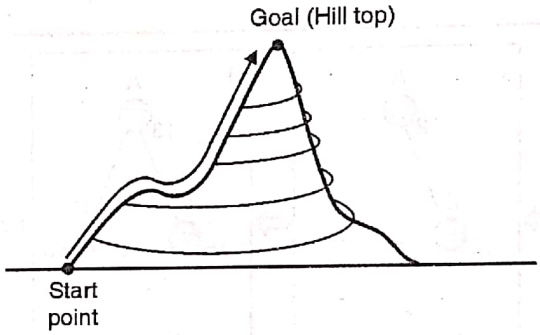


Fig. 2.17.1 : Hill Climbing Scenario

- Hill climbing always attempts to make changes that improve the current state. In other words, hill climbing can only advance if there is a higher point in the adjacent landscape.
- Hill climbing is a type of local search technique. It is relatively simple to implement. In many cases where state space is of moderate size, hill climbing works even better than many advanced techniques.
- For example, hill climbing when applied to travelling salesman problem; initially it produces random combinations of solutions having all the cities visited. Then it selects the better rout by switching the order, which visits all the cities in minimum cost.
- There are two variations of hill climbing as discussed follow.

2.17.1(A) Simple Hill Climbing

It is the simplest way to implement hill climbing. Following is the algorithm for simple hill climbing technique. Overall the procedure looks similar to that of generate and test but, the main difference between the two is use of heuristic function for state evaluation which is used in hill climbing. The goodness of any state is decided by the heuristic value of that state. It can be either incremental heuristic or detrimental one.

Algorithm

1. Evaluate the initial state. If it is a goal state, then return and quit; otherwise make it a current state and go to Step 2.
2. Loop until a solution is found or there are no new operators left to be applied (i.e. no new children nodes left to be explored).
   a. Select and apply a new operator (i.e. generate new child node)
   b. Evaluate the new state :
      (i) If it is a goal state, then return and quit.

(ii) If it is better than current state then make it a new current state.

(iii) If it is not better than the current state then continue the loop, go to Step 2.

As we study the algorithm, we observe that in every pass the first node / state that is better than the current state is considered for further exploration. This strategy may not guarantee that most optimal solution to the problem, but may save upon the execution time.

2.17.1(B) Steepest Ascent Hill Climbing

As the name suggests, steepest hill climbing always finds the steepest path to hill top. It does so by selecting the best node among all children of the current node / state. All the states are evaluated using heuristic function. Obviously, the time requirement of this strategy is more as compared to the previous one. The algorithm for steepest ascent hill climbing is as follows.

Algorithm

1. Evaluate the initial state, if it is a goal state, return and quit; otherwise make it as a current state.
2. Loop until a solution is found or a complete iteration produces no change to current state :
   a. SUCC = a state such that any possible successor of the current state will be better than SUCC.
   b. For each operator that applies to the current state, evaluate the new state :
      (i) If it is goal; then return and quit
      (ii) If it is better than SUCC then set SUCC to this state.
   c. SUCC is better than the current state → set the current state to SUCC.

- As we compare simple hill climbing with steepest ascent, we find that there is a tradeoff for the time requirement and the accuracy or optimality of the solution.
- In case of simple hill climbing technique as we go for first better successor, the time is saved as all the successors are not evaluated but it may lead to more number of nodes and branches getting explored, in turn the solution found may not be the optimal one.
- While in case of steepest ascent hill climbing technique, as every time the best among all the successors is selected for further expansion, it involves more time in evaluating all the successors at earlier stages, but the solution found will be always the optimal solution, as only the states leading to hill top are explored. This also makes it clear that the evaluation function i.e. the heuristic function definition plays a vital role in deciding the performance of the algorithm.

2.17.1(C) Limitations of Hill Climbing

MU -May 13, May 14, Dec. 14, May 15

Q. What are the limitations of Hill Climbing ? (May 13, Dec.14, 5 Marks)

Q. Write short note on Limitations of Hill-climbing algorithm. (May 14, May 15, 5 Marks)

- Now let's see what can be the impact of incorrect design of heuristic function on the hill climbing techniques.
- Following are the problems that may arise in hill climbing strategy. Sometimes the algorithms may lead to a position, which is not a solution, but from which there is no move possible which will lead to a better place on hill i.e. no further state that is going closer to the solution. This will happen if we have reached one of the following three states.

1. **Local Maximum :** A "local maximum" is a location in hill which is at height from other parts of the hill but is not the actual hill top. In the search tree, it is a state better than all its neighbors, but there is not next better state which can be chosen for further expansion. Local maximum sometimes occur within sight of a solution. In such cases they are called "Foothills".

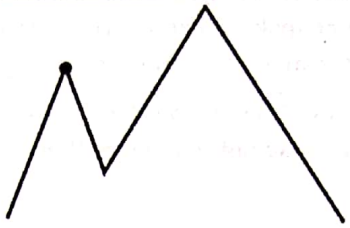


Fig. 2.17.2 : Local Maximum

- In the search tree local maximum can be seen as follows :

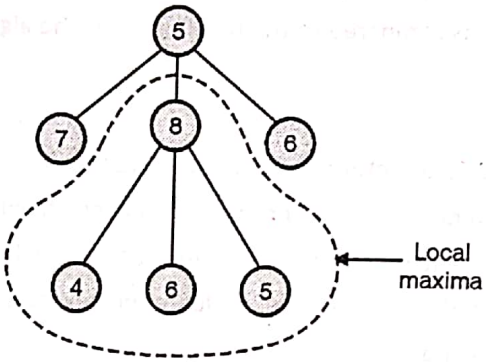


Fig. 2.17.3 : Local maxima in search tree

2. **Plateau :** A "plateau" is a flat area at some height in hilly region. There is a large area of same height in plateau. In the search space, plateau situation occurs when all the neighbouring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

Fig. 2.17.4 : Plateau in hill climbing

- In the search tree plateau can be identified as follows :

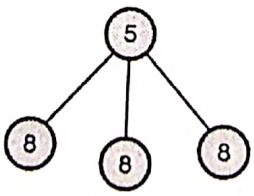


Fig. 2.17.5

3. **Ridge :** A "ridge" is an area in the hill such that, it is higher than the surrounding areas, but there is no further uphill path from ridge. In the search tree it is the situation, where all successors are either of same value or lesser, it's a ridge condition. The suitable successor cannot be searched in a simple move.

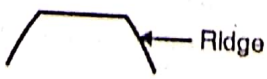


Fig. 2.17.6 : Ridge in hill climbing

- In the search tree ridge can be identified as follows :

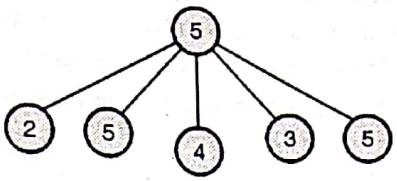


Fig. 2.17.7

- Fig. 2.17.8 depicts all the different situations together in hill climbing.

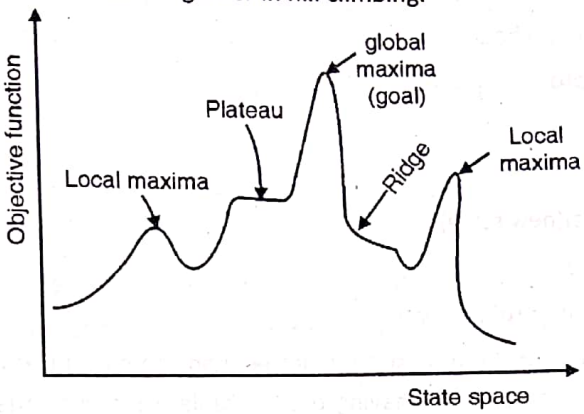


Fig. 2.17.8 : Hill climbing problems scenario

2.17.1(D) Solutions on Problems in Hill Climbing

- In order to overcome these problems we can try following techniques. At times combination of two techniques will provide a better solution.
  1. A good way to deal with local maximum, we can back track to some earlier nodes and try a different direction.
  2. In case of plateau and ridges, make a big jump in some direction to a new area in the search. This can be done by applying two more rules of the same rule several times, before testing. This is a good strategy is dealing with plateau and ridges.
- Hill climbing is a local method. It decides what to do next by looking only at the "immediate" consequences of its choices. Global information might be encoded in heuristic functions. Hill climbing becomes inefficient in large problem spaces, and when combinatorial explosion occurs. But it uses very little memory, usually a constant amount, as it doesn't retain the path.
- It is a useful when combined with other methods. The success of hill climbing depends very much on the shape of the state space. If there are few local maxima and plateau, random-restart hill climbing will find a good solution very quickly.

2.17.2 Simulated Annealing

- Simulated annealing is a variation of hill climbing. Simulated annealing technique can be explained by an analogy to annealing in solids. In the annealing process in case of solids, a solid is heated past melting point and then cooled.
- With the changing rate of cooling, the solid changes its properties. If the liquid is cooled slowly, it gets transformed in steady frozen state and forms crystals. While, if it is cooled quickly, the crystal formation will not get enough time and it produces imperfect crystals.
- The aim of physical annealing process is to produce a minimal energy final state after raising the substance to high energy level. Hence in simulated annealing we are actually going downhill and the heuristic function is a minimal heuristic. The final state is the one with minimum value, and rather than climbing up in this case we are descending the valley.

- The idea is to use simulated annealing to search for feasible solutions and converge to an optimal solution. In order to achieve that, at the beginning of the process, some downhill moves may be made. These downhill moves are made purposely, to do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state. It reduces the chances of getting caught at a local maximum, or plateau, or a ridge.

Algorithm

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied :

   Set T according to an annealing schedule

   Select and apply a new operator

   Evaluate the new state :

   goal→ quit

   ΔE = Val(current state) –Val(new state)

   $\Delta E < 0 \rightarrow$ new current state

   else→ new current state with probability $e^{-\Delta E/kT}$.

- We observe in the algorithm that, if the next state is better than the current, it readily accepts it as a new current state. But in case when the next state is not having the desirable value even then it accepts that state with some probability, $e^{-\Delta E/kT}$ where ΔE is the positive change in the energy level, T is temperature and k is Boltzmann's constant.
- Thus, in the simulated annealing there are very less chances of large uphill moves than the small one. Also, the probability of uphill moves decreases with the temperature decrease. Hence uphill moves are more likely in the beginning of the annealing process, when the temperature is high. As the cooling process starts, temperature comes down, in turn the uphill moves. Downhill moves are allowed any time in the whole process. In this way, comparatively very small upward moves are allowed till finally, the process converges to a local minimum configuration, i.e. the desired low point destination in the valley.

2.17.2(A) Comparing Simulated Annealing with Hill Climbing

- A hill climbing algorithm never makes "downhill" moves toward states with lower value and it can be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely **random walk**, i.e. moving to a successor chosen at random from the set of success or s independent of whether it is better than the current state, is complete but extremely inefficient. Therefore, it is reasonable to try a combination of hill climbing with a random walk in some way that yields both efficiency and completeness. Simulated annealing is the answer...!!
- As we know that, hill climbing can get stuck at local minima or maxima, thereby halting the algorithm abruptly, it may not guarantee optimal solution. Few attempts were made to solve this problem by trying hill climbing considering multiple start points or by increasing the size of neighbourhood, but none worked out to produce satisfactory results. Simulated annealing has solved the problem by performing some downhill moves at the beginning of search so that, local maximum can be avoided at later stage.
- Hill climbing procedure chooses the best state from those available or at least better than the current state for further expansion. Unlike hill climbing, simulated annealing chooses a random move from the neighbourhood. If the successor state turned out to be better than its current state then simulated annealing will accept it for further expansion. If the successor state is worse, then it will be accepted based on some probability.

2.17.3 Local Beam Search

- In all the variations of hill climbing till now, we have considered only one node getting selected at a time for further search process. These algorithms are memory efficient in that sense. But when an unfruitful branch gets explored even for some amount of time it is a complete waste of time and memory. Also the solution produce may not be the optimal one.
- The local beam search algorithm keeps track of k best states by performing parallel k searches. At each step it generates successor nodes and selects k best nodes for next level of search. Thus rather than focusing on only one branch it concentrates on k paths which seems to be promising. If any of the successors found to be the goal, search process stops.
- In parallel local beam search, the parallel threads communicate to each other, hence useful information is passed among the parallel search threads.
- In turn, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly terminates unfruitful branches exploration and moves its resources to where the path seems most promising. In stochastic beam search the maintained successor states are chosen with a probability based on their goodness.

Algorithm : Local Beam search

Step 1 : Found = false;

Step 2 : NODE = Root_node;

Step 3 : If NODE is the goal node, then *Found = true* else find SUCCs of NODE, if any with its estimated cost and store in OPEN list;

Step 4 : While (Found = false and not able to proceed further)

```
{
Sort OPEN list;
Select top W elements from OPEN list and put it in W_OPEN list and empty OPEN list;
   While (W_OPEN ≠ϕ and Found = false)
   {
   Get NODE from W_OPEN;
      if NODE = Goal state then Found = true else
      {
      Find SUCCs of NODE, if any with its estimated cost
      store in OPEN list;
      }
} // end inner while
} // end outer while
```

Step 5 : If *Found = true* then return *Yes* otherwise return *No* and Stop

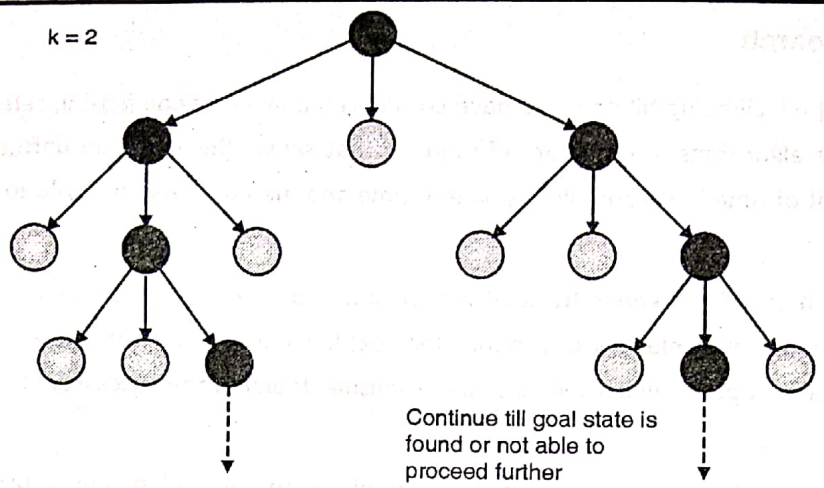


Fig. 2.17.9 : Process of local Beam Search

- As shown in Fig. 2.17.9, here k = 2, hence two better successors are selected for expansion at first level of search and at each next level, two better successors will be selected by both searches. They do exchange their information with each other throughout the search process. The search will continue till goal state is found or no further search is possible.
- It may seem to be that local beam search is same as running k searches in parallel. But it is not so. In case of parallel searches, all search run independent of each other. While in case of local beam search, the parallel running threads continuously coordinate with one another to decide the fruitful region of the search tree.
- Local beam search can suffer from a lack of diversity among the k states by quickly concentrating to small region of the state space.
- While it is possible to get excellent fits to training data, the application of back propagation is fraught with difficulties and pitfalls for the prediction of the performance on independent test data. Unlike most other learning systems that have been previously discussed, there are far more choices to be made in applying the gradient descent method.
- The key variations of these choices are : The learning rate and local minima - the selection of a learning rate is of critical importance in finding the true global minimum of the error distance.
- Back propagation training with too small a learning rate will make agonizingly slow progress. Too large a learning rate will proceed much faster, but may simply produce oscillations between relatively poor solutions.
- Both of these conditions are generally detectable through experimentation and sampling of results after a fixed number of training epochs.
- Typical values for the learning rate parameter are numbers between 0 and 1 : $0.05 < h < 0.75$.
- One would like to use the largest learning rate that still converges to the minimum solution momentum - empirical evidence shows that the use of a term called momentum in the back propagation algorithm can be helpful in speeding the convergence and avoiding local minima.
- The idea about using a momentum is to stabilize the weight change by making non radical revisions using a combination of the gradient decreasing term with a fraction of the previous weight change :

  $$\Delta w(t) = -\partial Ee/\partial w(t) + \alpha \Delta w(t-1)$$

  where a is taken 0£ a £ 0.9, and t is the index of the current weight change.
- This gives the system a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term.
- The momentum has the following effects :

- It smooths the weight changes and suppresses cross-stitching, that is cancels side-to-side oscillations across the error valey;
- When all weight changes are all in the same direction the momentum amplifies the learning rate causing a faster convergence;
- Enables to escape from small local minima on the error surface.

- The hope is that the momentum will allow a larger learning rate and that this will speed convergence and avoid local minima. On the other hand, a learning rate of 1 with no momentum will be much faster when no problem with local minima or non-convergence is encountered ;· sequential or random presentation - the epoch is the fundamental unit for training, and the length of training often is measured in terms of epochs. During a training epoch with revision after a particular example, the examples can be presented in the same sequential order, or the examples could be presented in a different random order for each epoch. The random representation usually yields better results.
- The randomness has advantages and disadvantages :
  - **Advantages :** It gives the algorithm some stochastic search properties. The weight state tends to jitter around its equilibrium, and may visit occasionally nearby points. Thus it may escape trapping in suboptimal weight configurations. The on-line learning may have a better chance of finding a global minimum than the true gradient descent technique.
  - **Disadvantages :** The weight vector never settles to a stable configuration. Having found a good minimum it may then continue to wander around it.
- Random initial state - unlike many other learning systems, the neural network begin in a random state. The network weights are initialized to some choice of random numbers with a range typically between -0.5 and 0.5 (The inputs are usually normalized to numbers between 0 and 1). Even with identical learning conditions, the random initial weights can lead to results that differ from one training session to another.
- The training sessions may be repeated till getting the best results.

2.17.4 Genetic Algorithms

MU – Dec. 15, May 16

Q. Write a short note on genetic algorithm.	**(Dec. 15, 8 Marks)**
Q. Explain how genetic algorithm can be used to solve a problem by tacking a suitable example.	**(May 16, 10 Marks)**

- Gas are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomized, Gas are by no means random, instead they exploit historical information to direct the search for better performance within the search space. The basic techniques of the Gas are designed to simulate processes in natural systems necessary for evolution, especially those following the principles of "survival of the fittest" laid down by Charles Darwin.
- Genetic algorithms are implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves towards better solutions. The solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individual and occurs in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified to form a new population. The new population is then used in the next iteration of the algorithm.

2.17.4(A) Terminologies of GA

- **Gene**

  Gene is the smallest unit in genetic algorithm. The gene represents the smallest unit of information in the problem domain and can be thought of as the basic building block for a possible solution. If the problem context were, for example, the creation of a well-balanced investment portfolio, a gene might represent the number of shares of a particular security to purchase.

- **Chromosome**

  Chromosome is a series of genes that represent the components of one possible solution to the problem. The chromosome is represented in computer memory as a bit string of binary digits that can be "decoded" by the genetic algorithm to determine how good a particular chromosome's gene pool solution is for a given problem. The decoding process simply informs the genetic algorithm what the various genes within the chromosome represent.

- **Encoding**

  Encoding of chromosomes is one of the problems, to start solving problem with GA. Encoding depends on the type of the problem. There are various types of encoding techniques like binary encoding, permutation encoding, value encoding, etc.

- **Population**

  A population is a pool of individuals (chromosomes) that will be sampled for selection and evaluation. The performance of each individual will be computed and a new population will be reproduced using standard genetic operators.

- **Reproduction**

  Reproduction is the process of creating new individuals called off-springs from the parents population. This new population will be evaluated again to select the desired results. Reproduction is done basically using two genetic operators : crossover and mutation. However, the genetic operators used can vary from model to model, there are a few standard or canonical operators : crossover and recombination of genetic material contained in different parent chromosomes, random mutation of data in individual chromosomes, and domain specific operations, such as migration of genes.

2.17.4(B) Genetic Operators

- **Selection**

  In this process, chromosomes are selected from the population to be the parents for the crossover. The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. There are many methods how to select the best chromosomes, for example roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection, etc.

- **Crossover**

  Crossover involves the exchange of gene information between two selected chromosomes. The purpose of the crossover operation is to allow the genetic algorithm to create new chromosomes that shares positive characteristics while simultaneously reducing the prevalence of negative characteristics in an otherwise reasonably fit solution. Types of crossover techniques include single-point crossover, two-point crossover, uniform crossover, mathematical crossover, tree crossover, etc.

Mutation

Mutation is another refinement step that randomly changes the value of a gene from its current setting to a completely different one. The majority of the mutations formed by this process are, as is often the case in nature, less fit than more so. Occasionally, however, a highly superior and beneficial mutation will occur. Mutation provides the genetic algorithm with the opportunity to create chromosomes and information genes that can explore previously uncharted areas of the solution space, thus increasing the chances for the discovery of an optimal solution. There are various types of mutation techniques like bit inversion, order changing, value encoding.

2.17.4(C) The Basic Genetic Algorithm

1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness f(x) of each chromosome x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete
4. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
5. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
6. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
7. [Accepting] Place new offspring in a new population
8. [Replace] Use new generated population for a further run of algorithm
9. [Test] If the end condition is satisfied, stop, and return the best solution in current population
10. [Loop] Go to step 2

2.17.4(D) Example of Genetic Algorithm

Let's consider following pair of chromosomes encoded using permutation encoding technique and are undergoing the complete process of GA. Assuming that they are selected using rank selection method and will be applied, arithmetic crossover and value encoding mutation techniques.

a. Parent chromosome

Chromosome A	153264798
Chromosome B	856723149

Child chromosome after **arithmetic crossover** : i.e adding bits of both chromosomes.

b. Child chromosome

Chromosome C	909 9 8 7837

After applying **value encoding mutation** i.e. adding or subtracting a small value to selected values. E.g. subtracting 1 to 3rd and 4th bit.

c. Child chromosome

Chromosome C	908 8 8 7837

It can be observed that the child produced is much better than both parents.

2.18 Adversarial Search

- **Adversarial Search Problem** is having competitive activity which involves 'n' players and it is played according to certain set of protocols.
- Game is called adversarial because there are agents with conflicting goals and the surrounding environment in a game is competitive as there are 'n' players or agents participating.
- We say that goals are conflicting and environment is competitive because every participant wants to win the game.
- From above explanation it is understandable that we are dealing with a competitive multi agent environment.
- As the actions of every agent are unpredictable there are many possible moves/actions.
- In order to play a game successfully every agent in environment has to first analyse the action of other agents and how other agents are contributing in its own wining or loosing. After performing this analysis agent executes.

2.18.1 Environment Types

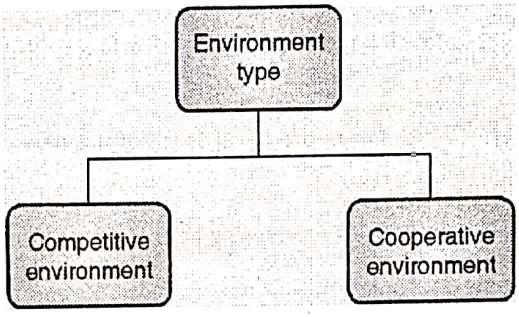


Fig. 2.18.1 : Environment types

There can be two types of environments in case of multi-agent : Competitive and cooperative.

1. **Competitive environment**
   - In this type of environment every agent makes an effort to win the game by defeating or by creating superior it over other agents who are also trying to win the game.
   - Chess is an example of a competitive environment.
2. **Cooperative environment**
   - In this type of environment all the agents jointly perform activities in order to achieve same goal.
   - Car driving agent is an example of a cooperative environment.

2.18.2 AI Game – Features

Under artificial intelligence category, there are few special features as shown in Table 2.18.1, which make the game more interesting.

Table 2.18.1 : AI game features

Features	Explanation	Example
Two player	When there are two opponents/agents playing the game it is called 2- play game. To increase difficulty of the game Intelligence is added to agents in AI games. Note that, in case of AI Games we must have at least two players. (i.e. single player games don't come under AI games category).	Chess

Features	Explanation	Example
Multi-agent	When there are two or more opponents/agents playing the game it is called multi agent environment where action of every agents affects the action of other agents.	Monopoly
Non-cooperative environment	When the surrounding environment is not helpful for winning the game it is called as non-cooperative or competitive.	Card games
Turn taking	In a multi-agent environment when the agent/ player performs a move and has to wait for the next player to make the next move.	Any board game, Chess, carom, etc.
Time limit	One more constraint can be incorporated in a game i.e. keeping a limitation on time. Every player will get a finite amount of time to take an action.	Time bound chess games
Unpredictable opponent	In AI games action of opponent agent is fuzzy which makes the game challenging and unpredictable. Players are called unpredictable when the next step depends upon an input set which is generated by the other player.	Card game with multiplayer

2.18.2(A) Zero Sum Game

- "Zero sum game" concept is associated with payoffs which are assigned to each player when the instance of the game is over. It is a mathematical representation of circumstances when the game is in a neutral state. (i.e. agents winning or losing is even handed with the winning and losing of other agents).
- For example, if player 1 wins chess game it is marked with say +1 point and at the same time the loss of player 2 is marked with – 1 point, thus sum is zero. Another condition is when game is draw; in that case players 1 and 2 are marked with zero points. (Here +1, – 1 and 0 are called payoffs).

2.18.2(B) Non-Zero Sum Game

Non-zero sum game's don't have algebraic sum of payoffs as zero. In this type of games one player winning the game does not necessarily mean that the other player has lost the game.

There are two types of non-zero sum games :

1. Positive sum game
2. Negative sum game

2.18.2(C) Positive Sum Game

It is also called as **cooperative game**. Here, all players have same goal and they contribute together to play the game. For example, educational games.

2.18.2(D) Negative Sum Game

It is also called as **competitive game**. Here, every player has a different goal so no one really wins the game, everybody loses. Real world example of a war suits the most.

2.19 Relevant Aspects of AI Game

To understand game playing, we will first take look at all appropriate aspects of a game which give overview of the stages in a game play. See Fig. 2.19.1.

- **Accessible environments :** Games with accessible environments have all the necessary information handy. For example : Chess.
- **Search :** Also there are games which require search functionality which illustrates how players have to search through possible game positions to play a game. For example : minesweeper, battleships.
- **Unpredictable opponent :** In AI games opponents can be unpredictable, this introduces uncertainty in game play and thus game-playing has to deal with contingency/ probability problems. For example : Scrabble.

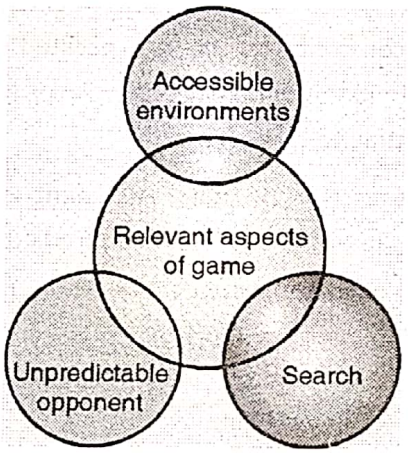


Fig. 2.19.1 : Relevant aspects of AI game

2.20 Game Playing

- Fig. 2.20.1shows examples of two main varieties of problems faced in artificial intelligence games. First type is "Toy Problems" and the other type is "Real World Problems".

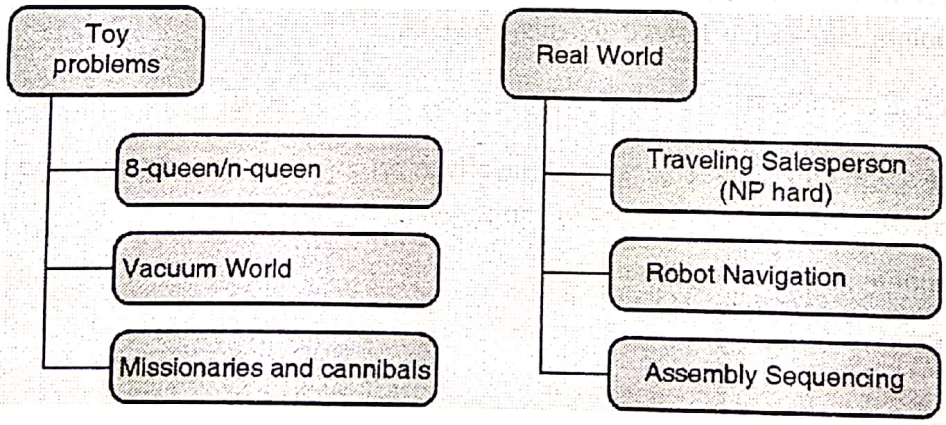


Fig. 2.20.1 : Example Problems

- Game play follows some strategies in order to mathematically analyse the game and generate possible outcomes. A two player strategy table can be seen in Table 2.20.1.

Table 2.20.1 : Two player strategy table

Player 1 / Player 2	Strategy 1	Strategy 2	Strategy 3	...
Strategy 1	Draw	Player 1 Wins	Player 2 Wins	...
Strategy 2	Player 1 Wins	Draw	Player 2 Wins	...
Strategy 3	Player 2 Wins	Player 2 Wins	Draw	...
□□□	□□□	□□□	□□□	...

2.20.1 Type of Game Strategies

- The simplest mathematical description of a game is the strategic form, mentioned in the introduction. For a two-person zero-sum game, the payoff function of Player II is the negative of the payoff of Player I, so we may restrict attention to the single payoff function of Player I, which we call here A.
- The strategic form, or normal form, of a two-person zero-sum game is given by a triplet (X, Y, A), where
  (1) X is a nonempty set, the set of strategies of Player I
  (2) Y is a nonempty set, the set of strategies of Player II
  (3) A is a real-valued function defined on $X \times Y$. (Thus, $A(x, y)$ is a real number for every $x \in X$ and every $y \in Y$.)
- The interpretation is as follows. Simultaneously, Player I chooses $x \in X$ and Player II chooses $y \in Y$, each unaware of the choice of the other. Then their choices are made known and I wins the amount $A(x, y)$ from II.
- This is a very simple definition of a game; yet it is broad enough to encompass the finite combinatorial games and games such as tic-tac-toe and chess.
- On the basis of how many times Player I or Player II is winning the game, following strategies can be discussed.

1. **Equalizing Strategy :** A strategy that produces the same average winnings no matter what the opponent does is called an **equalizing strategy.**
2. **Optimal Strategy :** If I has a procedure that guarantees him at least A(x, y) amount on the average, and II has a procedure that keeps her average loss to at most A(x, y). Then A(x, y) is called the value of the game, and the procedure each uses to insure this return is called an optimal strategy or a minimax strategy.
3. **Pure Strategies and Mixed Strategies :** It is useful to make a distinction between a pure strategy and a mixed strategy. We refer to elements of X or Y as pure strategies. The more complex entity that chooses among the pure strategies at random in various proportions is called a mixed strategy.

2.20.2 Type of Games

Game can be classified under deterministic or probabilistic category. Let's see what we mean by deterministic and Probabilistic.

(a) Deterministic

- It is a fully observable environment. When there are two agents playing the game alternatively and the final results of the game are equal and opposite then the game is called deterministic.
- Take example of tic-tac-toe where two players play a game alternatively and when one player wins a game then other player losses game.

(b) Probabilistic

- Probabilistic is also called as non-deterministic type. It is opposite of deterministic games, where you can have multiple players and you cannot determine the next action of the player.
- You can only predict the probability of the next action. To understand probabilistic type you can take example of card games.
- Another way of classification for games can be based on exact/perfect information or based on inexact / approximate information. Now, let us understand these terms.

1. **Exact/perfect information :** Games in which all the actions are known to other player is called as game of exact or perfect information. For example tic-tac-toe or board games like chess, checkers.
2. **Inexact / approximate information :** Game in which all the actions are not known to other players (or actions are unpredictable) is called game of inexact or approximate information. In this type of game, player's next action depends upon who played last, who won last hand, etc. For example card games like hearts.

Consider following games and see how they are classified into various types of games based on the parameters which we have learnt in above sections :

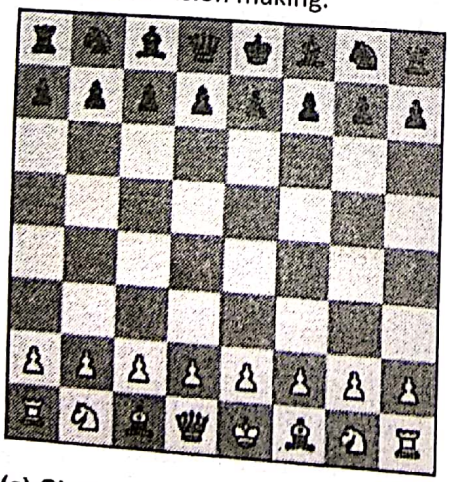
Table 2.20.2 : Types of game

	Exact/perfect information	Inexact / approximate information
Deterministic	– Chess – Checkers	– Battleships – Card Game (Hearts)
Probabilistic	– Ludo – Monopoly	– Scrabble – Poker

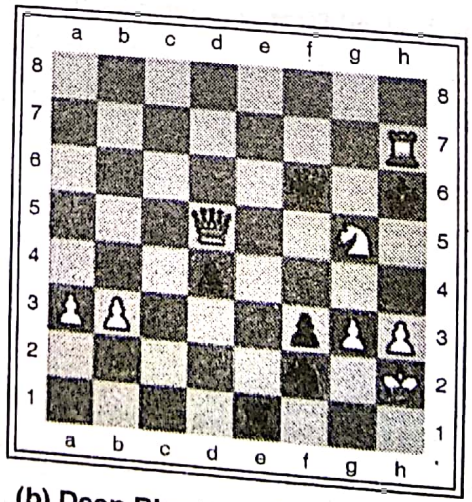
Now, let us try to learn about few games mention in Table 2.20.2.

2.20.2(A) Chess

- Chess comes under deterministic and exact/perfect information category. This game is a two person, zero-sum game.
- In chess both players can see chess board positions so there is no secrecy and players don't play at the same time they play one after the other in an order.
- Thus this game has perfect environment to test artificial intelligence techniques. In 1990's a computer Deep Blue II defeated Garry Kasparov who was world champion at that time. This example is given to understand how artificial intelligence can be used in decision making.



(a) Chess board



(b) Deep Blue II vs Garry Kasparov final position game 1

Fig. 2.20.2

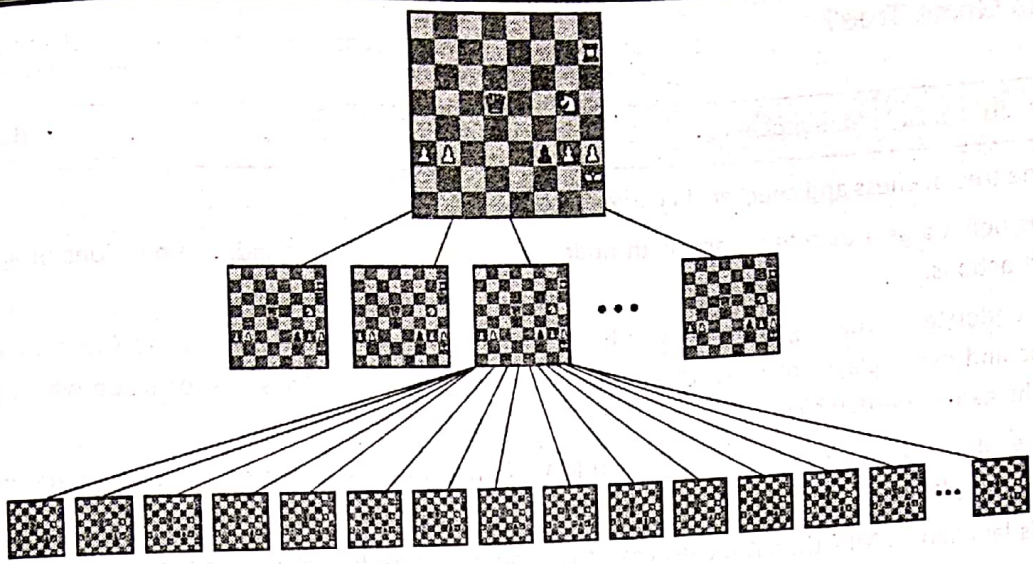


Fig. 2.20.3 : Chess game tree

2.20.2(B) Checkers

- Checkers comes under deterministic and exact/perfect information category. This game is a two person game where both players can see board positions, so there is no secrecy and players play one after the other in an order.
- In 1990's a computer program name Chinook (also called draughts) was developed which defeated human world champion Marion Tinsley.

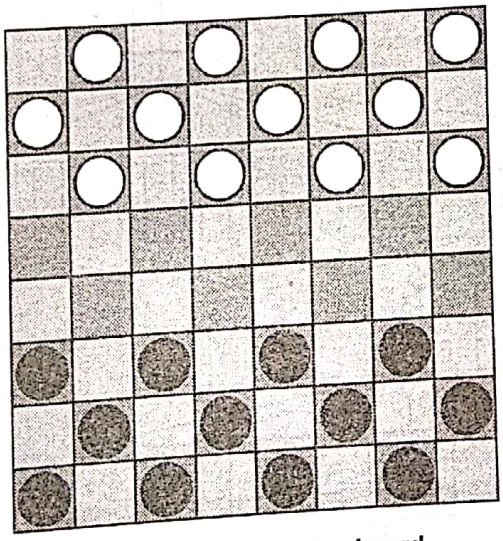


Fig. 2.20.4 : Checkers board

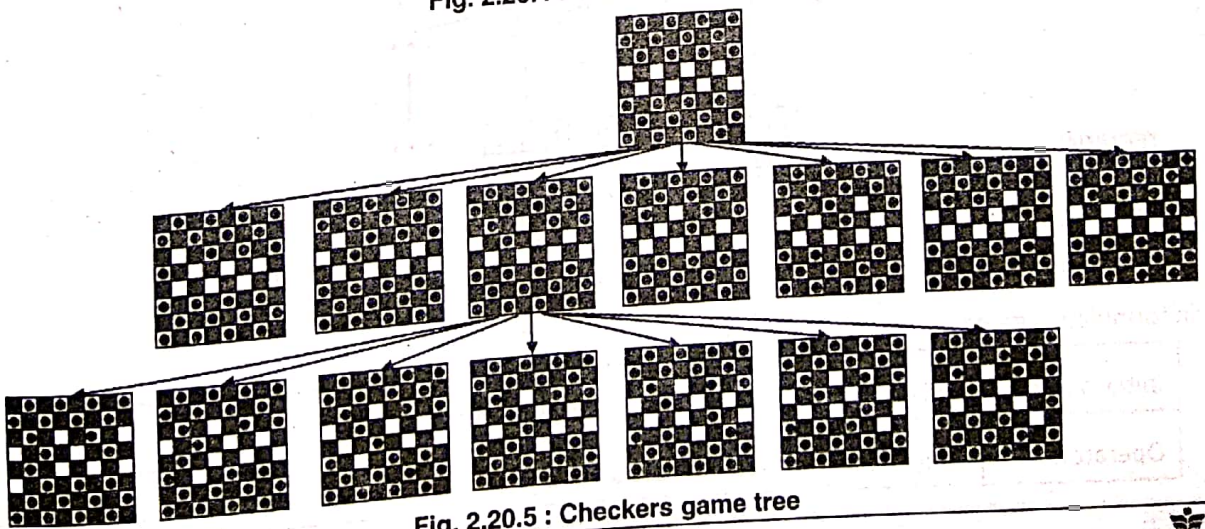


Fig. 2.20.5 : Checkers game tree

2.20.3 What is Game Tree?

MU – Dec. 15

Q. Draw game tree of tic-tac-toe problem. **(Dec. 15, 5 Marks)**

- We saw game tree of chess and checkers in earlier section.
- Game tree is defined as a directed graph with nodes and edge. Here nodes indicate positions in a game and edges indicate next actions.
- Let us try to understand what a game tree is with the help of Tic-Tac-Toe example. Tic-Tac-Toe is a 2-player game, it is deterministic and every player plays when his/her turn come. Game tree has a **Root Node** which give the starting position of the game board, which is a blank 3×3 grid.
- Say in given example player 1 takes 'X' sign. Then MAX(X) indicates the board for the best single next move. Also it indicates that it is player 1's turn. (Remember that initial move is always indicated with a MAX).
- If the node is labelled as MIN then it means that it is opponents turn (i.e. player 2's turn) and the board for the best single next move is shown.
- Possible moves are represented with the help of lines. Last level shows terminal board position, which illustrates end of the game instance, Here, we get zero as sum of all the play offs. Terminal state gives winning board position. Utility indicates the play off points gained by the player (– 1, 0, + 1).
- In a similar way we can draw a game tree for any artificial intelligence based games.

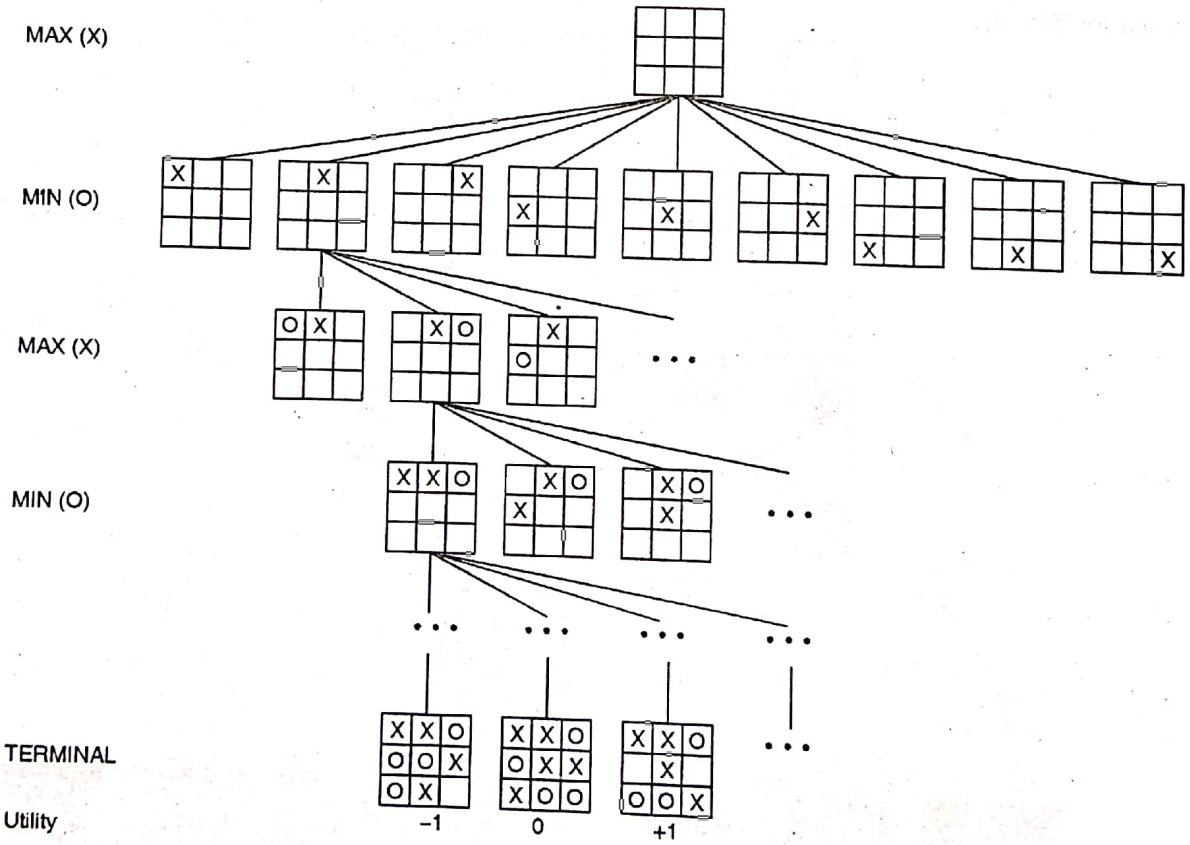


Fig. 2.20.3 : Tic-tac-toe game tree

We can formulate a game as a search problem as follows :

Initial state	It gives the starting position of the game board.
Operators	They define the next possible actions.

Terminal state	Which indicates that all instance of game are over.
Utility	It displays a number which indicates if the game was won or lost or it was draw.

- From Tic-Tac-Toe game's example you can understand that for a 3 × 3 grid, two player game, where the game tree is relatively small (It has 9! terminal nodes), still we cannot draw it completely on one single page.
- Imagine how difficult it would be to create a game tree for multi-player games or for the games with bigger grid size.
- Many games have huge search space complexity. Games have limitation over the time and the amount of memory space it can consume. Finding an optimal solution is not feasible most of the times, so there is a need for approximation.
- Therefore there is a need for an algorithm which will reduce the tree size and eventually will help in reducing the processing time and in saving memory space of the machine.
- One method is **pruning** where, only the required parts, which improve quality of output, of the tree are kept and reaming parts are removed.
- Another method is **heuristic method** (it makes use of an **evaluation function)** it does not require exhaustive research. This method depends upon readily available information which can be used to control problem solving.

2.21 MiniMax Algorithm

- Minimax algorithm evaluates decision based on the present status of the game. This algorithm needs deterministic environment with perfect/exact information.
- Minimax algorithm directly implements the defining equation. Every time based on the successor state minimax value is calculated with the help of simple recursive computation.
- In case of minimax algorithm the selected action with highest minimax value should be equal to the best possible payoff (outcome) against best play.

2.21.1 Minimax Algorithm

Take example of tic-tac toe game to understand minimax algorithm. We will take a random stage.

Step 1 : Create an entire game tree including all the terminal states.

Start action : 'O'

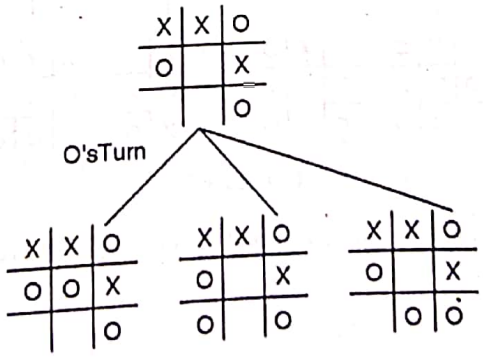


Fig. 2.21.1

Next action : 'X'

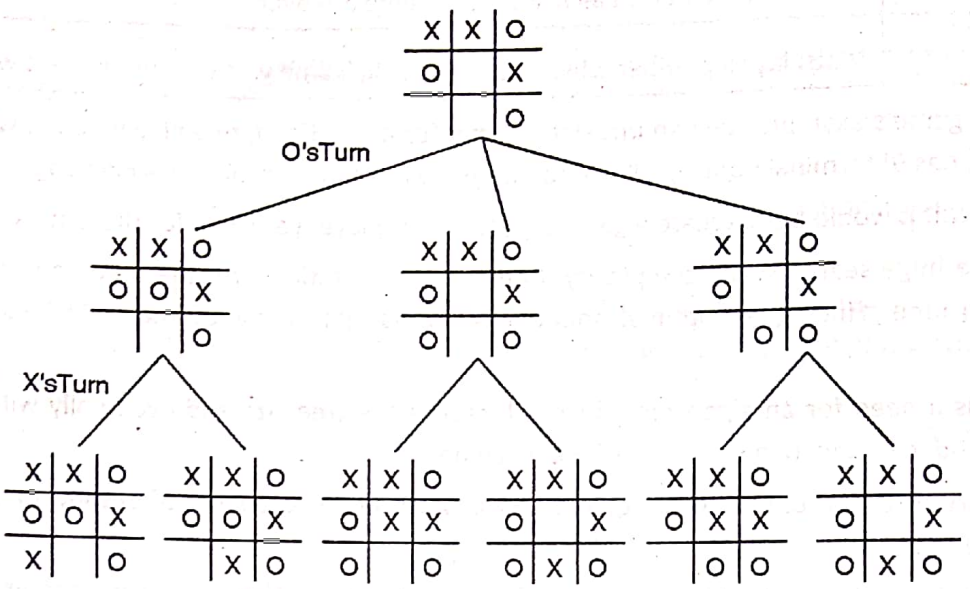


Fig. 2.21.2

Next action : 'O'

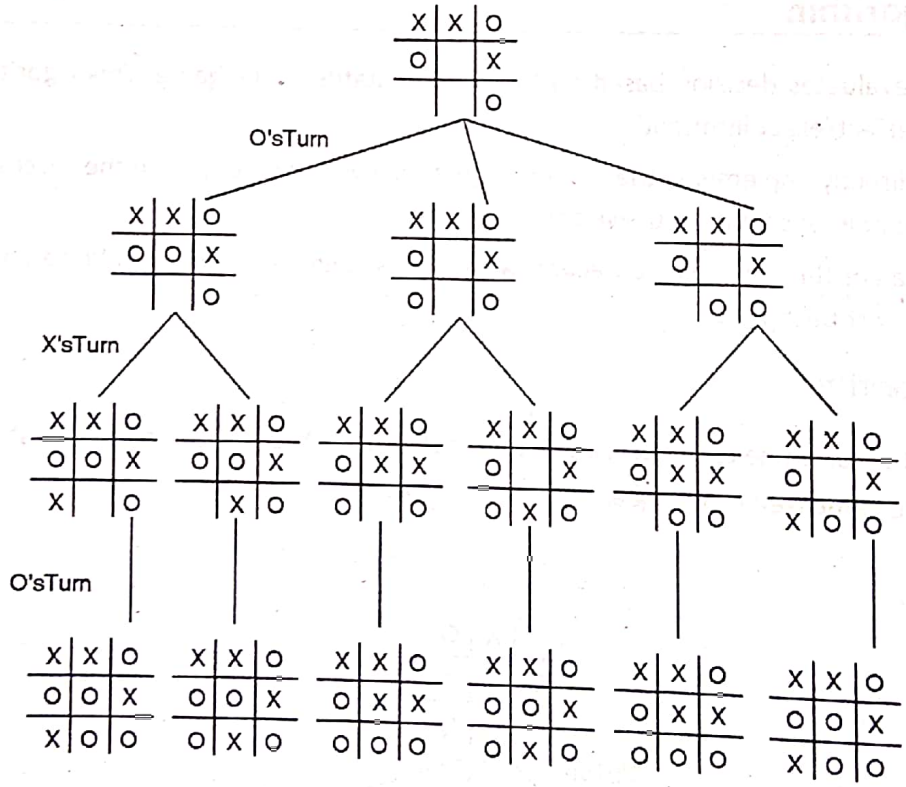


Fig. 2.21.3

Step 2 : For every terminal state find out utility (playoff points gained by every terminal state). Terminal position where 1 means win and 0 means draw.

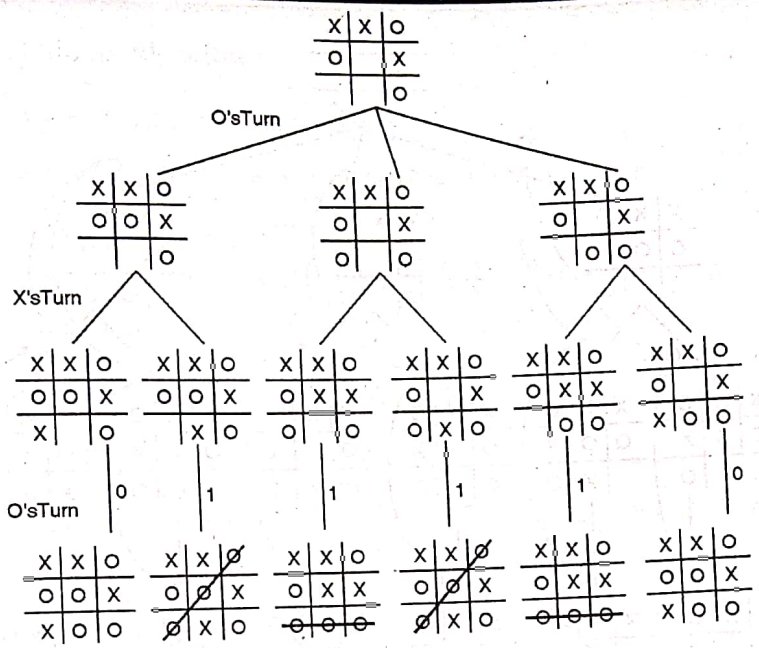


Fig. 2.21.4

Step 3 : Apply MIN and MAX operators on the nodes of the present stage and propagate the utility values upward in the three.

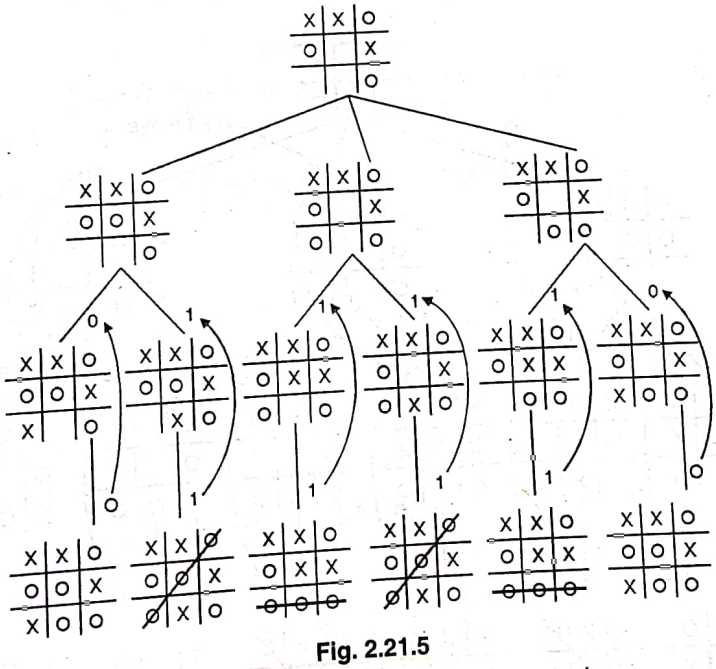


Fig. 2.21.5

Step 4 : With the max (of the min) utility value (payoff value) select the action at the root node using minimax decision.

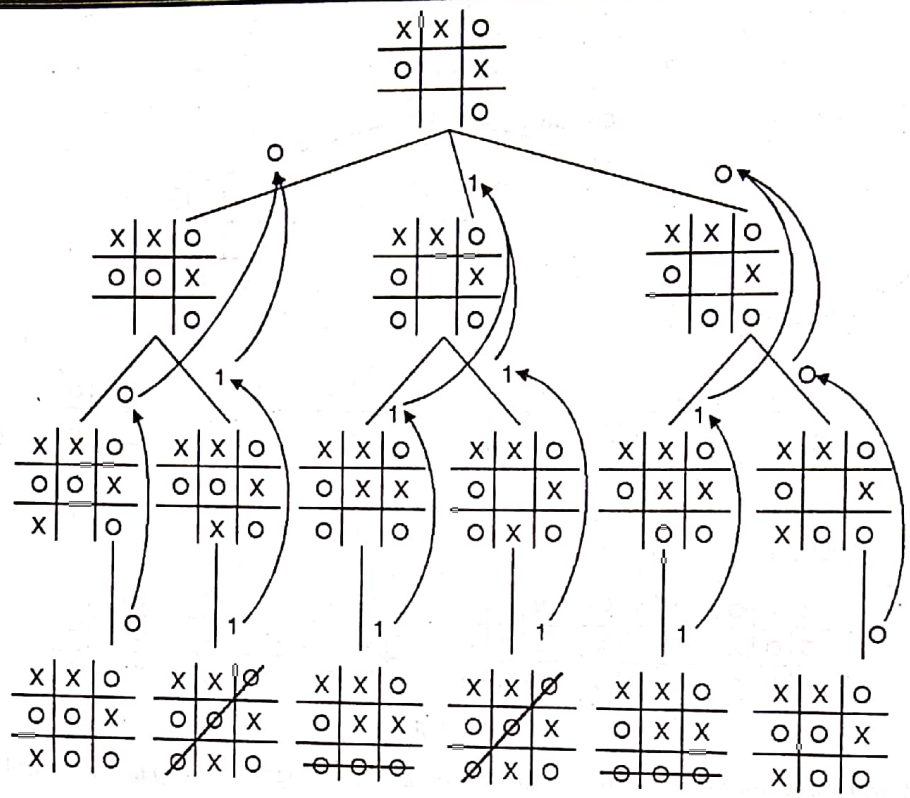


Fig. 2.21.6

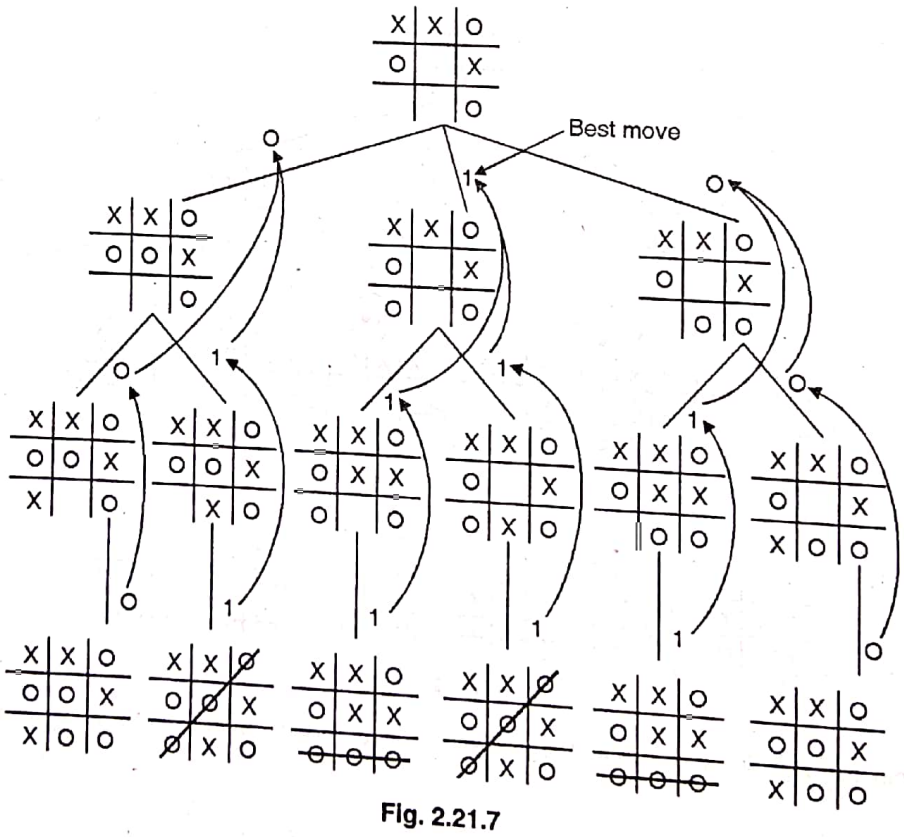


Fig. 2.21.7

(In case of Steps 2 and 3 we are assuming that the opponent will play perfectly as per our expectation)

2.21.2 Properties of Minimax Algorithm

- It is considered as Complete if the game tree size is finite.
- It is considered Optimal when it is played against an optimal number of opponents.
- Time complexity of minimax algorithm is indicated as $O(b^m)$.
- Space complexity of minimax algorithm is indicated by $O(b^m)$(using depth-first exploration approach).
- For chess, b ≈ 35, m ≈100 for "reasonable" games.
- Exact solution is completely infeasible in most of the games.

2.22 Alpha Beta Pruning

- Pruning means cutting off. In game search it resembles to clipping a branch in the search tree, probably which is not so fruitful.
- At any choice point along the path for max, α is considered as the value of the best possible choice found i.e., highest-value. For each "X", if "X" is worse i.e. lesser value than α value then, MAX will avoid it. Similarly we can define β value for MIN.
- α-β pruning is an extension to minimax algorithm where, decision making process need not consider each and every node of the game tree.
- Only the important nodes for quality output are considered in decision making. Pruning helps in making the search more efficient.
- **Pruning** keeps only those parts of the tree which contribute in improving the quality of the result remaining parts of the tree are removed.
- Consider the following game tree :

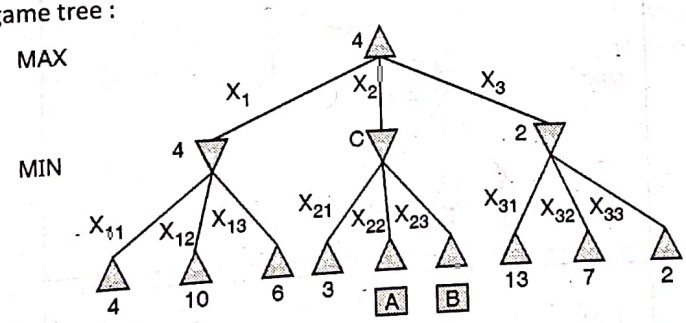


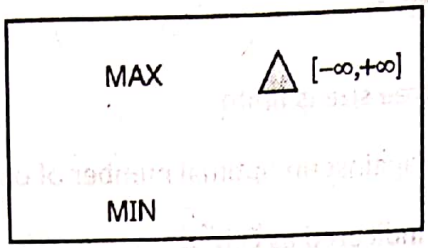
Fig. 2.22.1

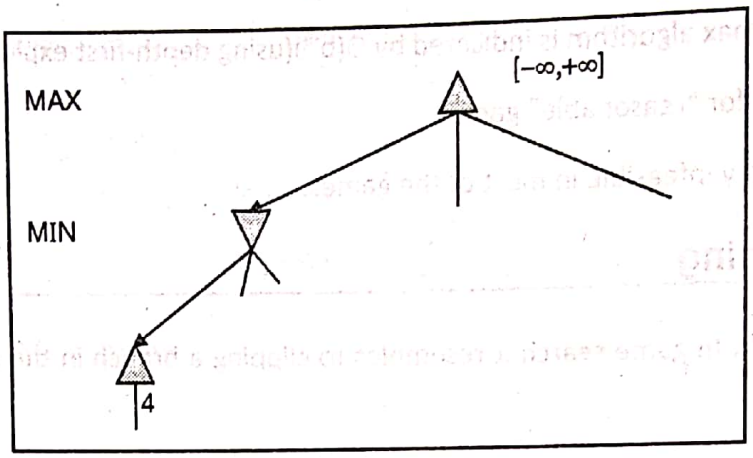
- For which we have to calculate minimax values for root node.

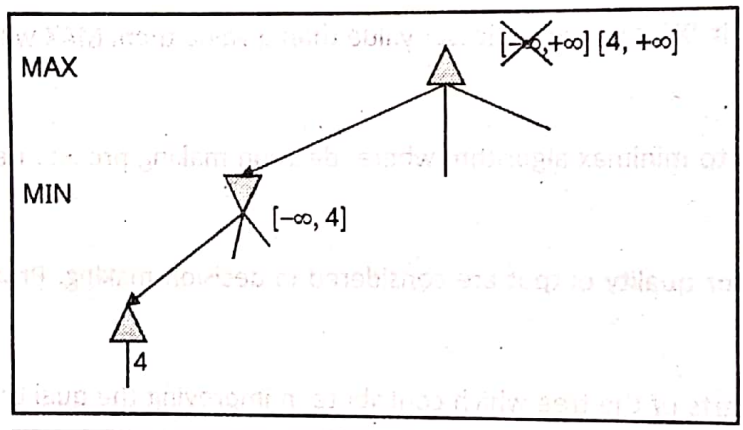
Minimax value of root node = Max(min(4,10,6), min(3,A,B), min(13,7,2))

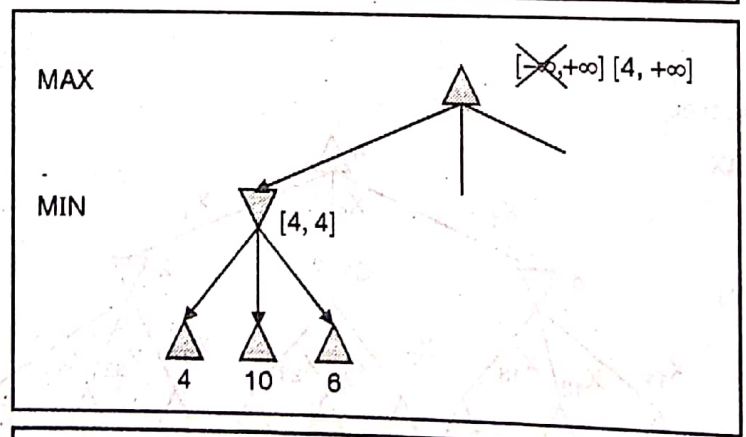
= Max(4, C, 2) (C <=3)=4

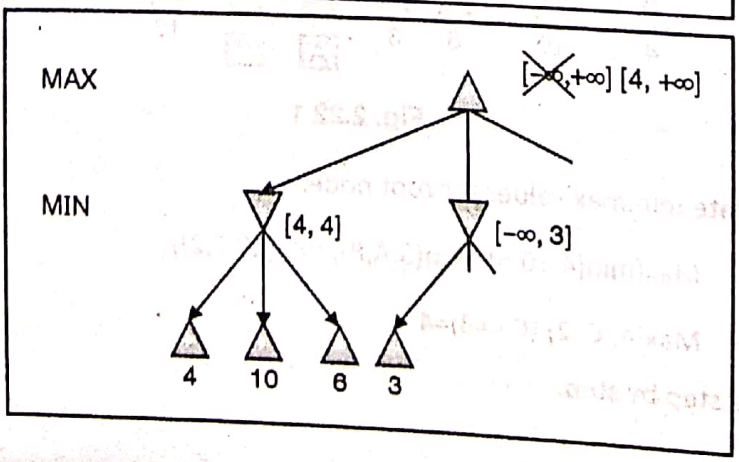
- Let us see how to check this step by step.

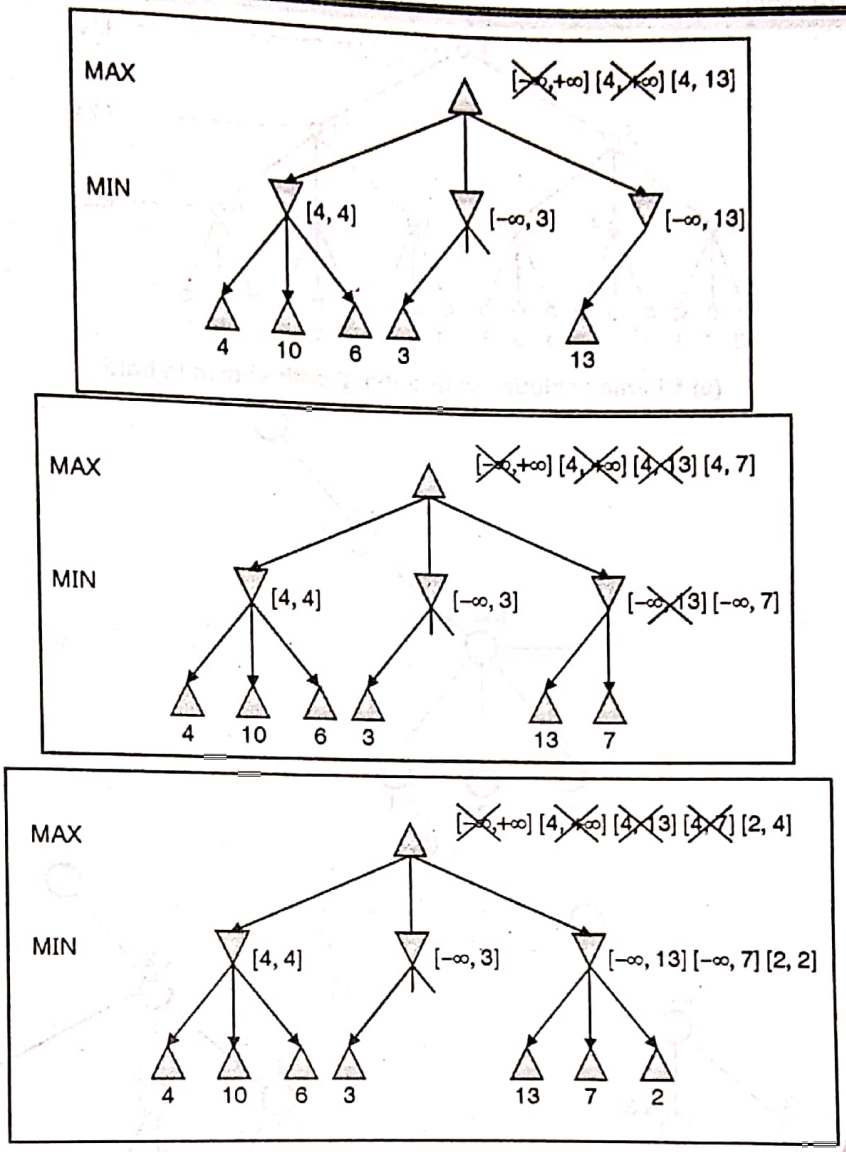












- So in this example we have pruned 2 β and 0 α branches. As the tree is very small, you may not appreciate the effect of branch pruning; but as we consider any real game tree, pruning creates a significant impact on search as far as the time and space is concern.

2.22.1 Example of α-β Pruning

Ex. 2.22.1 : Explain Min-Max and Alpha Beta pruning algorithm with following example.

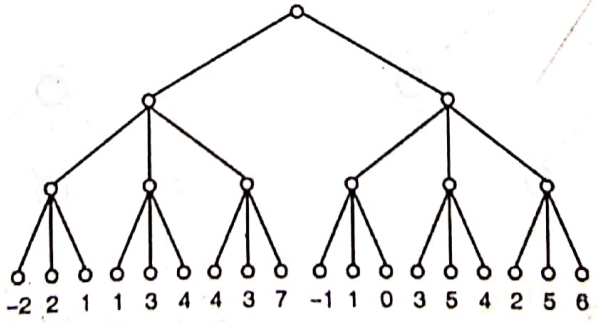
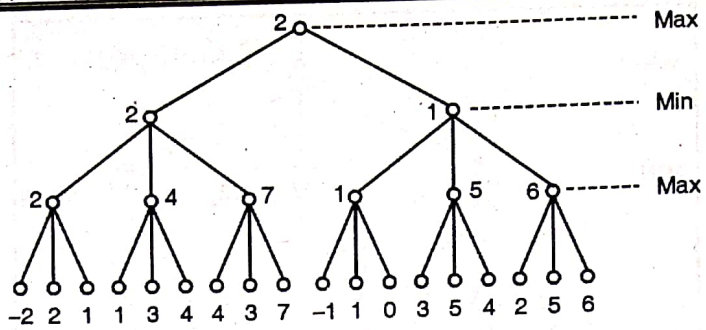
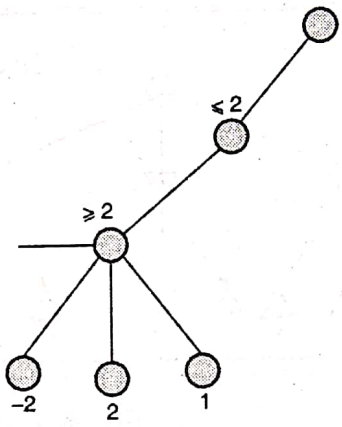


Fig. P. 2.22.1 : Game Tree

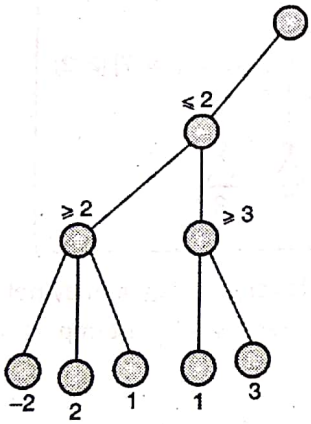
Soln. : Using min-max algorithm



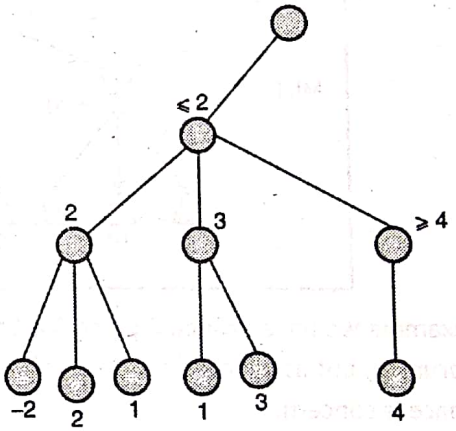
(a) Min-max solution with optimal path shown in bold



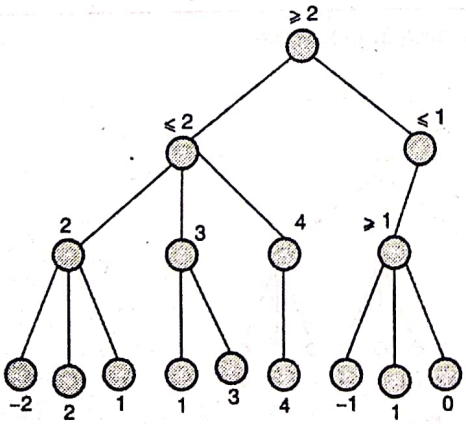
(b)



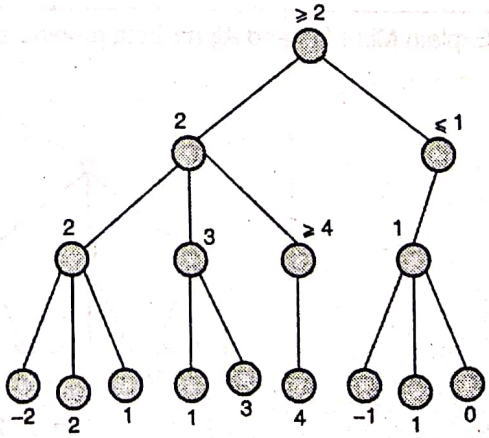
(c)



(d)

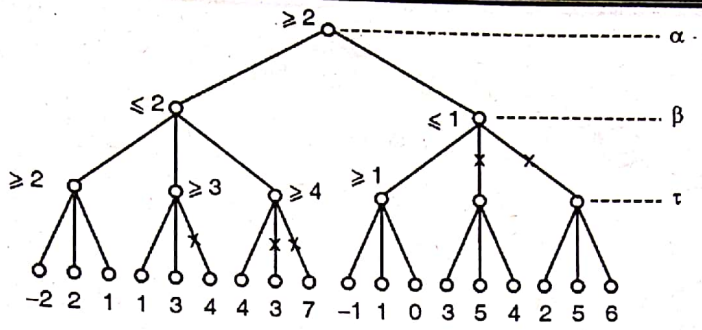


(e)



(f)

Fig. P. 2.22.1



(g) Solution by α-β pruning

Fig. P. 2.22.1

Total pruned branches

α - cuts = 2

β - cuts = 3

Ex. 2.22.2 : Perform α-β cutoff on the following.

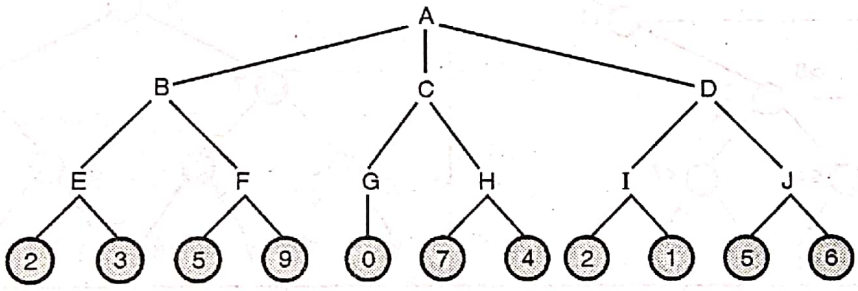


Fig. P. 2.22.2

Soln. :

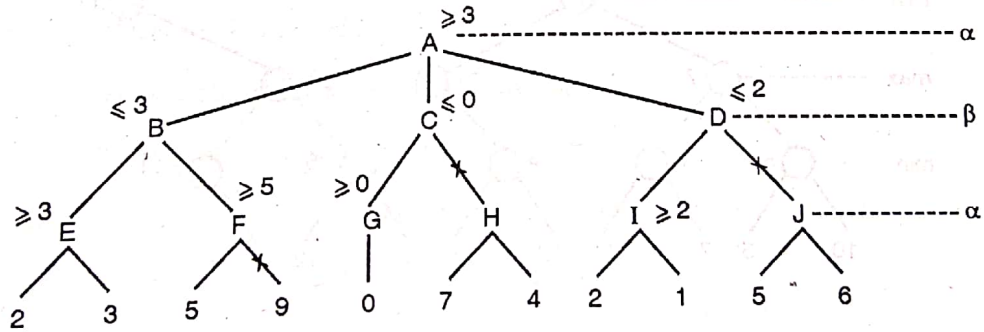


Fig. P. 2.22.2(a)

No. of α - cuts = 1

No. of β - cuts = 2

Ex. 2.22.3 : Apply alpha-beta pruning on example given in Fig. P.2.22.3 considering first node as max.

MU - May 16, 10 Marks

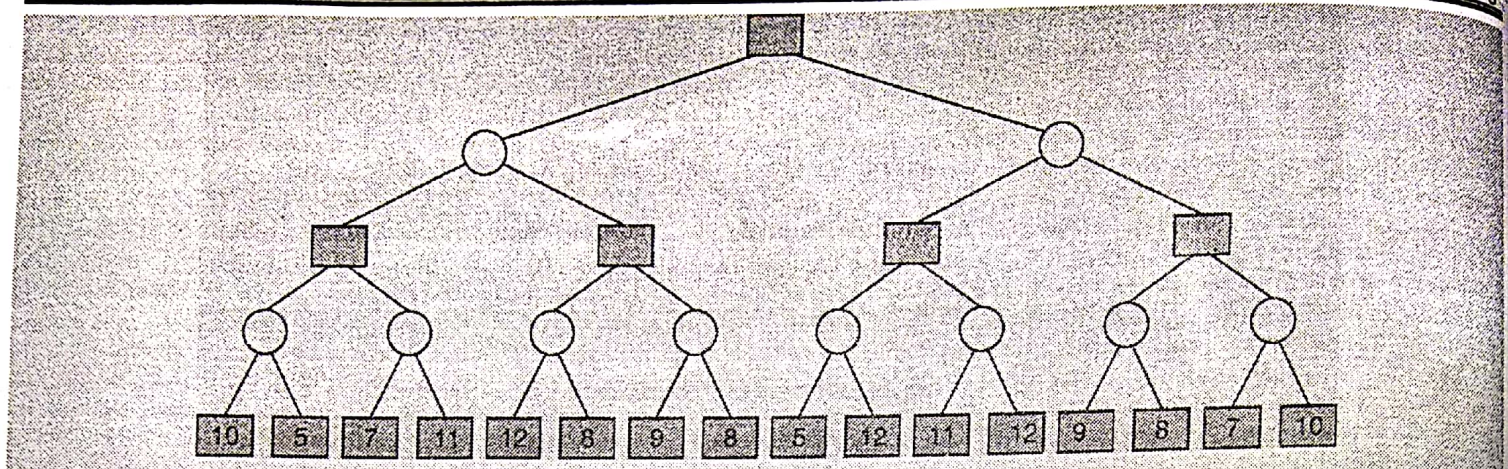
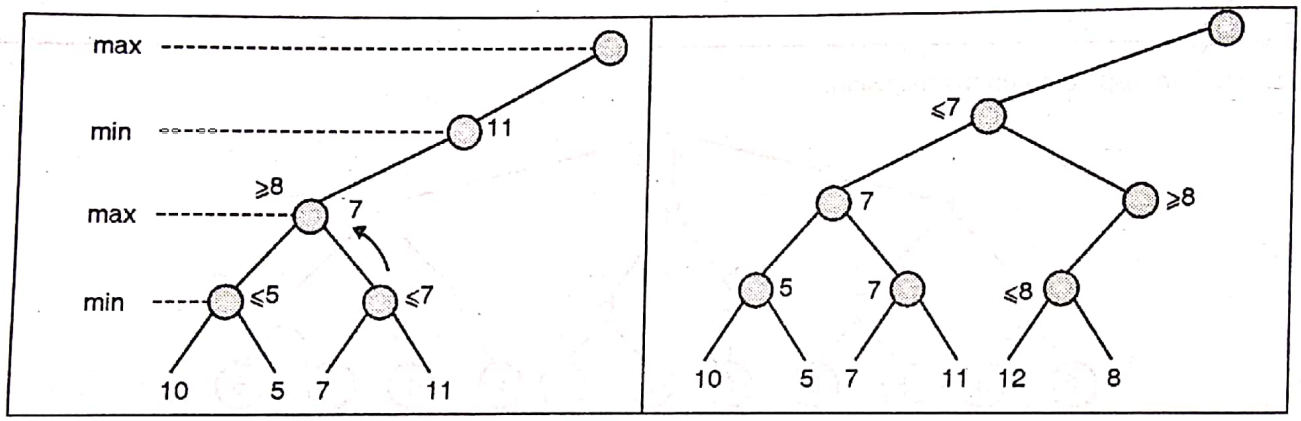
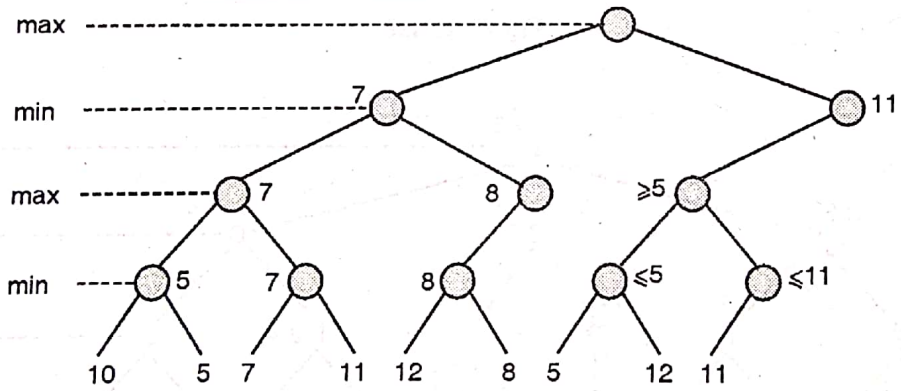
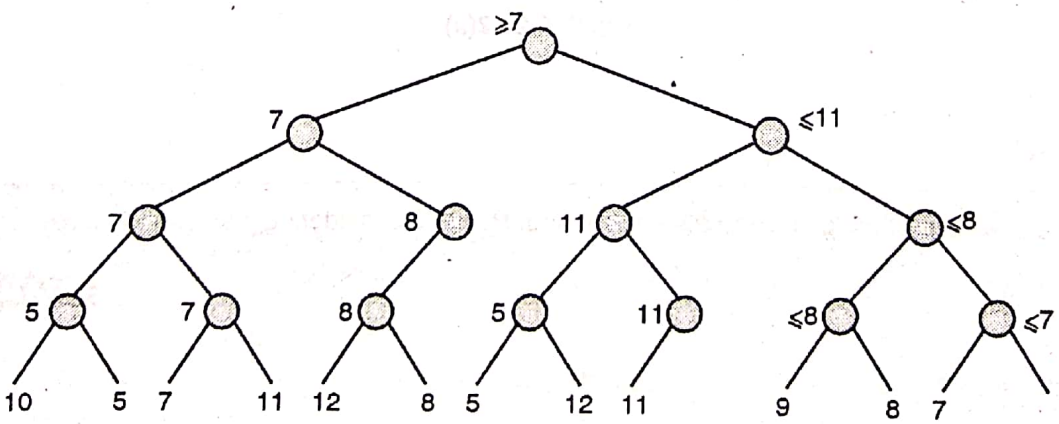


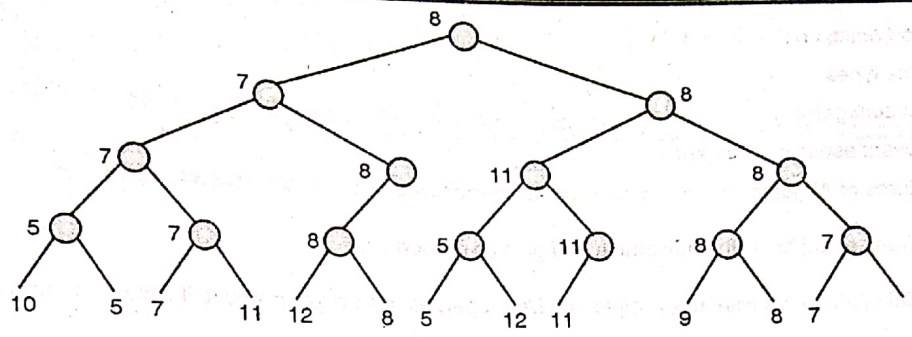
Fig. P.2.22.3

Soln. :









2.22.3 Properties of α-β

- Final results are not affected by pruning.
- Ordering of Good actions helps in improving effectiveness of pruning technique.
- If there is exact/perfect ordering then we can get time complexity as $O(b^{m/2})$.
- Depth of search is doubled with pruning.

Review Questions

Q.1 Differentiate between BFS and DFS.

Q. 2 How the drawbacks of DFS are overcome by DLS and DFID?

Q. 3 Compare and contrast all the un-informed searching techniques.

Q. 4 Write short note on bidirectional search.

Q. 5 Write a note on BFA and Uniform cost search.

Q. 6 Compare and contrast DFS, FLS and DFID.

Q. 7 What are various informed search techniques? Explain A* with example.

Q. 8 Compare Best First Search and A* with an example.

Q. 9 Write algorithm of steepest ascent hill climbing. And compare it with simple hill climbing.

Q. 10 What are the limitations of hill climbing? How can we solve them?

Q. 11 Write algorithm for Best first search and specify its properties.

Q. 12 What is the difference between best first and greedy best first search? Explain with example.

Q. 13 What is heuristic function? What are the qualities of a good heuristic?

Q. 14 Write short note on simulated annealing and local beam search.

Q. 15 Compare and contrast Simulated annealing with Hill climbing.

Q. 16 How the definition of heuristic affects the search process? explain with suitable example.

Q. 17 Write short note on behavior of A* in case of underestimating and overestimating Heuristic.

Q. 18 Discuss admissibility of A* in case of optimality.

Q. 19 Explain SMA* algorithm with example. When should we choose SMA* given options?

Q. 20 Write a short notes on :

(a) Game types

(b) Zero-sum game

(c) Relevant aspects of AI games

(d) Features of AI game

Q. 21 Explain minimax algorithm with an example and give its properties.

Q. 22 Give α-β pruning algorithm with an example and it's properties, also explain why is it called α-β pruning.

Q. 23 What is adversarial search?

Q. 24 Apply alpha-beta pruning on example given in Fig. Q. 24 considering first node as max.

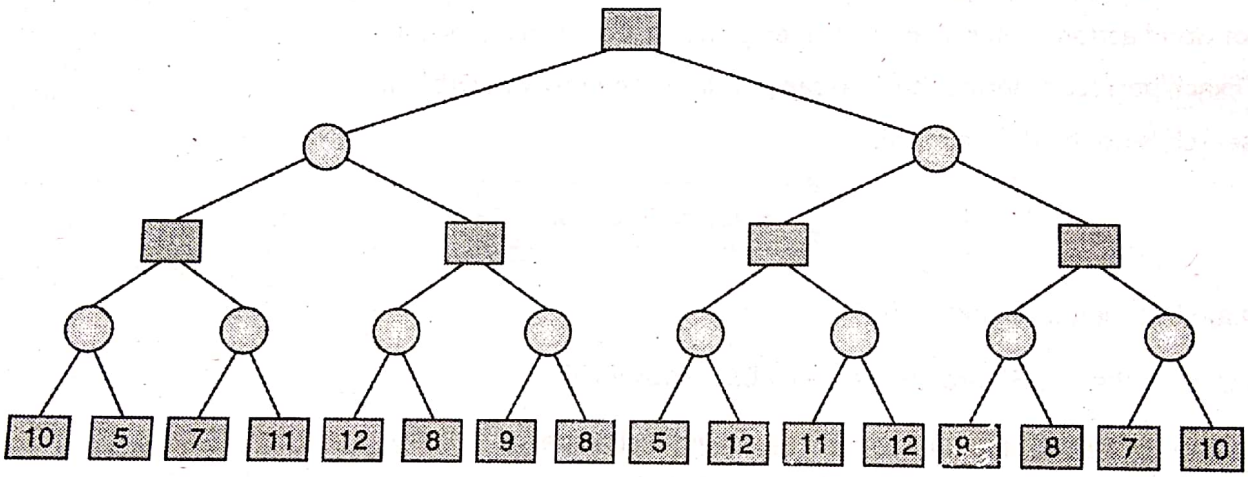


Fig. Q. 24