Module 3

CHAPTER 3

Problem Solving

Syllabus

Uninformed Search Methods : Breadth First Search (BFS), Depth First Search (DFS), Depth Limited Search, Depth First Iterative Deepening (DFID), Informed Search Methods: Greedy best first Search, A* Search, Memory bounded heuristic Search.

Local Search Algorithms and Optimization Problems: Hill climbing search Simulated annealing, Genetic algorithms.

Adversarial Search: Game Playing, Min-Max Search, Alpha Beta Pruning

Module 3

3.1 PROBLEM SOLVING

GQ. Briefly explain problem solving.

Problem solving consists of using generic or ad hoc methods, in an orderly manner, for finding solutions to problems. Some of the problem-solving techniques use artificial intelligence, computer science, engineering, mathematics, or psychoanalysis.

Problems can also be classified into two different types: **ill-defined and well-defined,** from which appropriate solutions are to be made.

1. **Ill-defined** problems are those that **do not have clear goals**, solution paths, or expected solution.
2. **Well-defined** problems **have specific goals**, clearly defined solution paths, and clear expected solutions. These problems also allow for more initial planning than ill-defined problems.

   Being able to solve problems sometimes involves dealing with **pragmatics (logic)** and **semantics** (interpretation of the problem). It requires the ability to understand what the goal of the problem is and what rules could be applied to represent the key to solving the problem. Sometimes the problem requires some abstract thinking and coming up with a creative solution.

Problem types

1. **Deterministic, fully observable** ⇒ single-state problem

   Agent knows exactly which state the problem will be in; it's solution is a sequence.
2. **Non-observable** ⇒ conformant problem

   Agent may have no idea where it is; solution (if any) is a sequence.
3. Nondeterministic and/or partially observable ⇒ contingency problem percepts provide new information about current state.
4. **Unknown state space** ⇒ exploration problem ("online") Solution is a tree or policy often interrelated search, and execution.

3.1.1 Problem Solving Agent

GQ. Describe problem solving agent.

Problem-solving agents : A goal formulation, based on the current situation and the performance measure is required for problem solving.

- **Problem formulation** is the process of deciding what actions and states to consider, given a goal. In general, an agent with several options for action of unknown value can decide what to do by first examining different possible sequences of actions (that lead to states of known value) and then choosing the best sequence.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence.

3.1.2 Simple Problem-Solving Agent

```
Function SIMPLE-PROBLEM-SOLVING- AGENT (percept)
returns an action Persistent:
        seq. an action sequence, initially empty,
        state, some description of the current world state,
        goal; a goal initially null,
        problem, a problem formulation,
        state ← UPDATE-STATE(state percept)
    If seq. is empty then
        goal ← FORMULATE-GOAL(State)
        problem ← FORMULATE-PROBLEM
                (State, goal)
        seq. ← SEARCH( problem}
    If seq. = failure then return a null action action
← FIRST(seq.)
    seq. ← REST(seq.) return action
```

3.2 PERFORMANCE MEASURES

- Performance measurement is the process of collecting, analysing and / or reporting information regarding the performance of an individual, group, organization, system or component.
- Moulin defines the term with a forward looking organizational focus 'the process of evaluating how well organizations are managed and the value they deliver for the customers and other stakeholders'.
- The most common such frameworks include :
  - (i) **Balanced scorecard :** It is used by organisations to manage the implementation of corporate strategies.

(ii) **Key performance indicator :** It is a method for choosing important / critical performance measures, usually in an organisational content.

- Performance measure algorithms are as follows :
  1. **Time complexity :** It is a measure of amount of time for an algorithm to execute.
  2. **Space efficiency :** It is a measure of the amount of memory needed for an algorithm to execute.
  3. **Complexity theory :** It is a study of algorithm performance.
  4. **Function dominance :** It is a comparison of cost functions.
  5. Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- It means that when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.
- To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.
- To analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.
- Based on this information, performance analysis of an algorithm can be defined as 'Performance analysis of an algorithm is the process of calculating space and time required by that algorithm'.
- Performance analysis of an algorithm is performed by using the following measures :
  (i) **Space complexity :** Space required to complete the task of that algorithm. It includes program space and data space.
  (ii) **Time complexity :** Time required to complete the task of that algorithm.
- This involves the following steps :
  (a) Implement the algorithm completely.
  (b) Determine the time required for each basic operation.
  (c) Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.

3.2.1 Types of Performance Measures

(a) Workload or output measures. These measures indicate the amount of work performed or number of services received.
(b) Efficiency measures.
(c) Effectiveness or outcome measures.
(d) Productivity measures.

3.3 SEARCHING

GQ. What is searching?

Search plays a major role in solving many artificial intelligence (AI) problems. Search is a **universal problem-solving mechanism** in AI. In many problems, sequence of steps required to solve a problem is not known in advance but must be determined by systematic trial-and-error exploration of alternatives.

Search techniques try to "pre-play" the game by evaluating the future states (game tree search) and may use also heuristic to prone bad choices or speed things up. They theoretically can make an **exact and perfect** choice, but are **slow**.

The problems that are addressed by AI search algorithms fall into three general- classes:

1. Single-agent path-finding problems
2. Two players games
3. Constraint- satisfaction problems

3.3.1 Node Representation in Search Tree

A binary search tree (BST) is a tree in which all the nodes follow the below mentioned properties

(i) The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
(ii) The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and it can be defined as left-sub-tree (keys) < node (key) ≤ right_sub-tree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and has an associated value.

Module 3

While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

We mention below a pictorial representation of BST

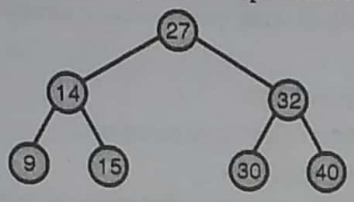


Fig. 3.3.1

Note that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

☞ Basic Operation

The basic operations of a tree

(i) Search : Searches an element in a tree

(ii) Insert : Inserts an element in a tree

(iii) Pre-order Traversal : Traverses a tree in a pre-order manner

(iv) In-order Traversal : Traverses a tree in an in-order manner

(v) Post-order Traversal : Traverse a tree in a post-order manner.

(vi) Remark : There must be no duplicate nodes.

☞ Nodes representing in search tree

Remark

Let us suppose we want to search for the number :

(i) We start at the root

(ii) We compare the value to be searched with the value of the root.

(iii) It is equal, we complete the search.

(iv) If it is lesser, we need to go to the left sub-tree, since in a binary subtree all the elements in the left subtree are lesser and all the elements in the right subtree are greater.

(v) In this traversal, at each step we discard one of the sub-trees.

(vi) We go on reducing like this till we find the element or till our search is reduced to only one node.

(vii) The search here is a binary search and hence is called as Binary search tree.

Note : After reaching the end, just insert that node at left (it less than current), else right.

⏭ 3.4 UNINFORMED SEARCH (BLIND SEARCH)

GQ. What is blind or uninformed search ?

UQ. Explain with example various uninformed search techniques. **(MU - Q. 3(a), May 17, 10 Marks)**

(1) They have no additional information about states other than those provided in the problem definition. They can only **generate successors and distinguish between goal state and non- goal state**.

(2) These are commonly used search procedures which explore all the alternating options during the search process. They do not have any dome in specific knowledge. All they need are the initial state, the final state and a set of legal operators.

Most important search techniques are as follows :

1. Depth first search.
2. Breadth first search.
3. Uniform-cost search.
4. Depth-limited search.
5. Iterative deepening search.
6. Bidirectional search.

⏭ 3.5 DEPTH FIRST SEARCH (DFS)

(1) DFS search is the **distributive file system**. The distributive file function provides the ability to logically group shares on multiple servers and to transparently link shares into a single namespace. DFS organizes shared resources on a network in a tree-like structure.

(2) DFS is a file-system with data stored on a server. The data is accessed and processed on if it was stored on the local client machine.

(3) The DFS makes it convenient to share information and files among users on a network in a controlled and authorised way.

(4) DFS is used in topological **sorting, scheduling problems, cycle detection in graphs and solving puzzles**, other applications involve analysing networks **e.g.** testing, if a graph is bipartite

(5) Depth First Search (DFS) is an algorithm for traversing or searching tree or graph data structures.

(6) The algorithm starts at the root node (in case of a graph, selecting some arbitrary node as the root node) and explores as far as possible along each branch before backtracking. Refer Fig. 3.5.1.

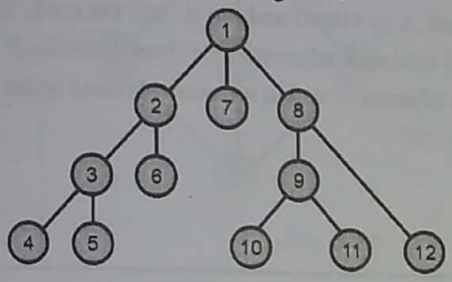


Fig. 3.5.1 : Depth First Search (DFS) order in which the nodes are visited

3.5.1 DFS Algorithm

DFS algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Refer Fig. 3.5.2. In this example, DFS **algorithm** traverses from S to A to D to G to E to B first, then to F and lastly to C. It follows the following rules :

- **Rule 1 :** Visit the adjacent unvisited vertex. Mark it as visited and display it.
- **Rule 2 :** If no adjacent vertex is found, choose the vertex from the stack. (There will appear all the vertices from the stack, which do not have adjacent vertices).
- **Rule 3 :** Repeat Rule 1 and Rule 2 till the stack becomes empty.

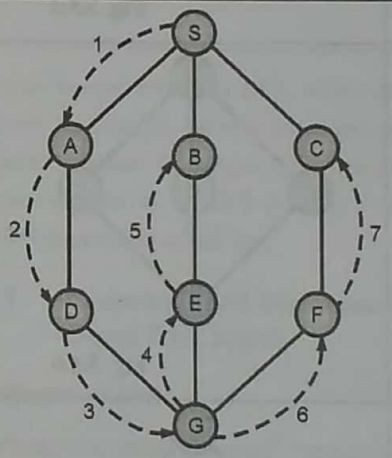
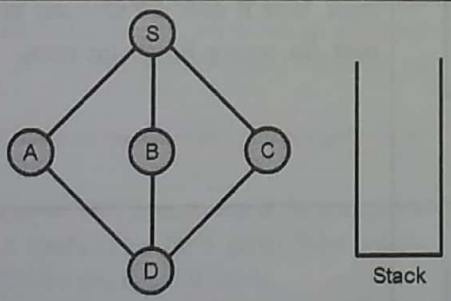
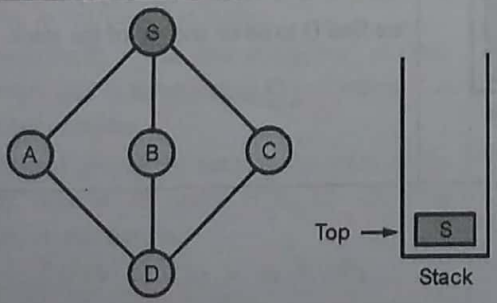
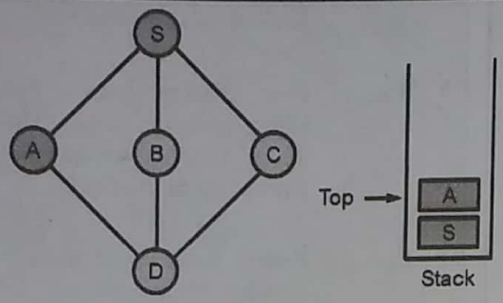
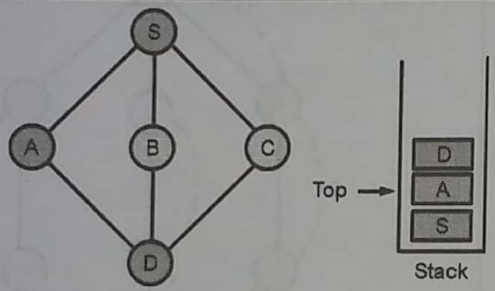
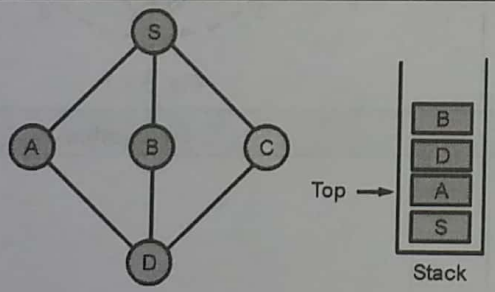
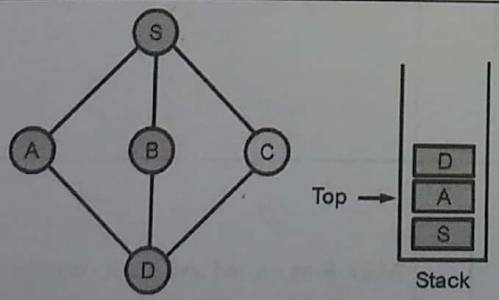
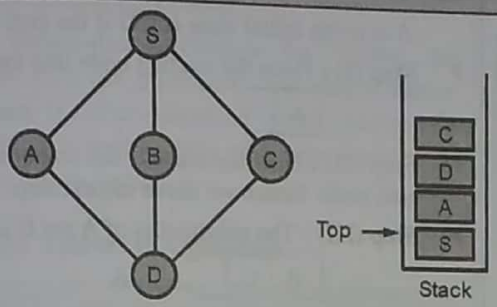


Fig. 3.5.2

Step	Traversal	Description
1.	 **Fig. 3.5.3**	Initialise the stack.
2.	 **Fig. 3.5.4**	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. Here we take the node in alphabetical order.

Step	Traversal	Description
3.	 Fig. 3.5.5	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Bothe S and D are adjacent to A, but we want unvisited nodes only.
4.	 Fig. 3.5.6	Visit D and mark it as visited and put onto the stack. Here we have B and C nodes, which are adjacent to D and both are unvisited. But again we choose in an alphabetical order.
5.	 Fig. 3.5.7	We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.	 Fig. 3.5.8	We check the stack top for return to the previous node and check if it has any unvisited nodes. Here we find D to be on the top of the stack.

Step	Traversal	Description
7.	 Fig. 3.5.9	Only unvisited adjacent node is from D is C. So we visit C, mark it and put it into the stack.

- As C does not have any unvisited adjacent node, so we keep popping the stack till we find a node that has an unvisited adjacent node. Here, there is none and the stack is empty and the program is over.

3.5.2 Performance Measures of DFS

The performance measuring factors of an algorithm are as follows :

- The two most common measures are : speed and memory usage; other measures could include transmission speed, temporary disk usage, long-term disk usage, power consumption, total cost of ownership, response time to external stimuli, etc.
- Performance measure is generally defined as regular measurement of outcomes and results which generates reliable data on the effectiveness and efficiency of programs.
- There are four ways to measure the performance of an algorithm :
  - (i) **Completeness :** DFS is complete if the search tree is finite, it implies that for a given finite search, DFS will have a solution if it exists.
  - (ii) **Optimality :** DFS is not optimal, it means that the number of steps in reaching the solution, or the cost spent in reaching it is high.
  - (iii) **Time complexity :** The time complexity of DFS, if the entire tree is traversed, is O (V) where V is the number of nodes.

    For a directed graph, the sum of the sizes of the adjacency lists of all nodes is E. So, the time complexity in this case is

    $$O(V) + O(E) = O(V + E)$$

    For an undirected graph, each edge appears twice.
  - (iv) **Space complexity :** For DFS, which goes along a single 'branch' all the way down and uses a stack implementation, the height of the tree matters. The space complexity for DFS is O (h) where h is the maximum height of the tree.

3.5.3 Advantages and Disadvantages of Depth First Search

Advantages of DFS

1. Memory requirement is linear with respect to nodes.
2. Less time and space complexity rather than BFS.
3. Solution can be found out without much more search.

Disadvantages of DFS

1. Not guarantee that it will give you solution.
2. Cut-off depth is smaller so time complexity is more.
3. Determination of depth until the search proceeds.
4. The major drawback of depth-first search is the determination of the depth till which the search is reached. This depth is called cut-off depth. The value of **cut-off depth** is essential because otherwise the search will go on and on.

   If the cut-off depth is smaller, solution may not be found and if cut-off depth is large, time-complexity will be more.

3.5.4 Applications of DFS

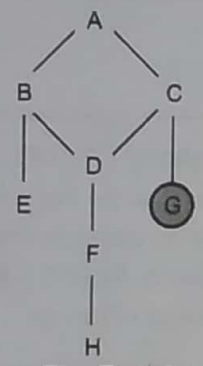
1. Finding connected components.
2. Topological sorting.
3. Finding bridges of graph.

Module 3

3.5.5 Solved Example on DFS

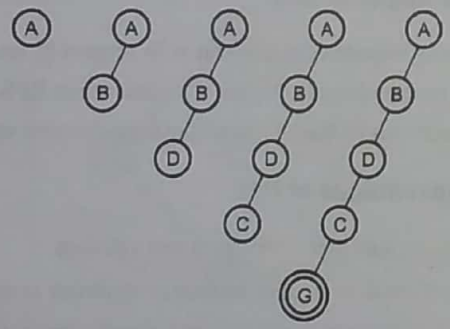
UEx. 3.5.1 (MU - Q. 2(b), May 18, 10 Marks. Q. 2(a), Dec. 15, 10 Marks)

Consider the following graph shown in Fig. 3.5.1 starting from 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.



(1B13)**Fig. Ex. 3.5.1**

Soln. :



(1B14)**Fig. Ex. 3.5.1(a)**

UEx. 3.5.2 : (MU-Q. 5(b), May 16, 10 Marks)

Consider the graph given in the figure. Assume that the initial state is A and the goal state is G. Find a path from the initial state to the goal state using DFS. Also report the solution cost.

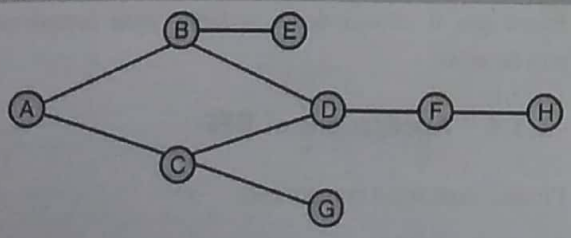


Fig. Ex. 3.5.2

Soln. :

A is given initial state and G is the goal node.

- **Step (I) :** Place the starting node into the stack

A	

- **Step (II) :** Now the stack is not empty and A is not our goal node. Hence we move to next step.
- **Step (III) :** The neighbours of A are B and C.

B	C		A

- **Step (IV) :** Now, B is top node of the stack. Its neighbours are E and D.

E	D		B

- **Step (V) :** E is top node of the stack. We find its neighbour, but there is no neighbour of E in the graph, so

B	D		E

- **Step (VI) :** Now, D is top node; and its neighbours are F and C; we push D into the stack.

F	C		D

- **Step (VII) :** Now, G is our top node of the stack, which is our goal node.
- **Step (VIII) :** solution cost is

$A \rightarrow B \rightarrow E \rightarrow D \rightarrow G$

3.6 BREADTH-FIRST SEARCH (BFS)

GQ. Comment upon statement that breadth-first search is a special case of uniform cost search.

GQ. Explain breadth first search with its algorithm.

(1) BFS stands for Breadth-First Search is a vertex based technique for finding a shortest path in graph. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

(2) BFS is the core of many graph analysis algorithms and it is used in many problems, such as social network, computer network analysis and data organization.

(3) BFS involves search through a tree one level at a time. We traverse through one entire level of children nodes first, before moving onto traverse through the grand children nodes.

(4) Breadth first search is an algorithm for node that satisfies a given property. It starts at the **tree root** and

explores all nodes at the present depth prior to moving on to nodes at the next depth level.

(5) BFS uses **Queue-data structure** for finding the shortest path. BFS can be used to find **single source shortest path** in an un-weighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.

3.6.1 BFS Traversal Algorithm

- **Step 1 :** Add a node / vertex from the graph to a queue of nodes to be 'visited'.,
  ↓
- **Step 2 :** Visit the topmost node in the queue, and mark it as such.
  ↓
- **Step 3 :** If that node has any neighbours, check to see if they have been 'visited' or not.
  ↓
- **Step 4 :** Add any neighbouring nodes that still need to be 'visited' to the queue.

Illustrative Example

Ex. 3.6.1 : Which solution would BFS find to move from node S to node G if run on the graph below ?

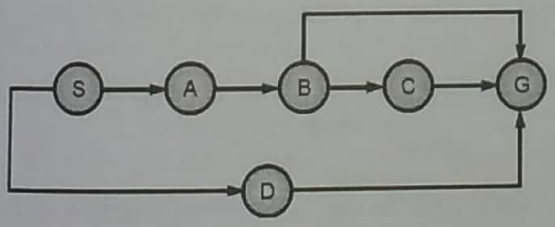


Fig. Ex. 3.6.1

Soln. :

The equivalent search tree for the above graph is as follows :

As BFS traverses the tree "shallowest node first", it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown by dotted line.

Path : S → D → G

= the depth of the shallowest solution.

= number of nodes in level.

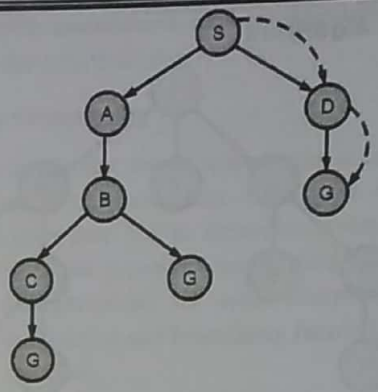


Fig. Ex. 3.6.1(a)

3.6.2 Performance Measures for BFS

Time complexity

Equivalent to the number of nodes traversed in BFS until the shallowest solution.

Note : $T(n) = 1 + n^2 + n^3 + ... + n^s = O(n^s)$.

Space complexity

Equivalent to how large can the fringe get :

$$S(n) = O(n^s)$$

Completeness

BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.

Optimality

BFS is optimal as long as the costs of all edges are equal.

3.6.3 BFS Algorithm for Extra Memory

- BFS is an algorithm for searching a tree data structure for a node that satisfies a given property.
- It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
- Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

Module 3

☞ **Sorting algorithm**

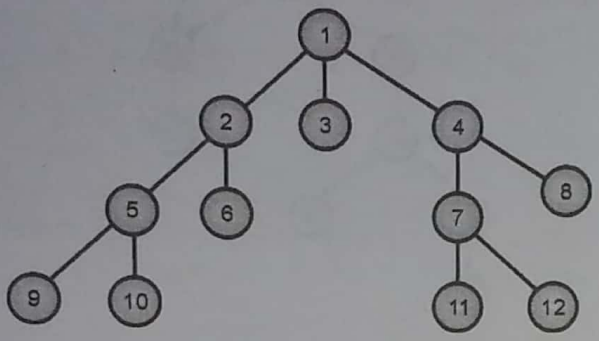


Fig. 3.6.1 : Order in which nodes are expanded

☞ **Concept diagram**

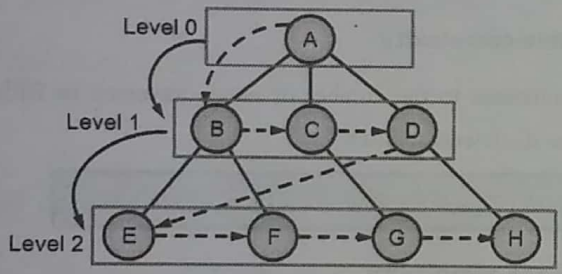


Fig. 3.6.2

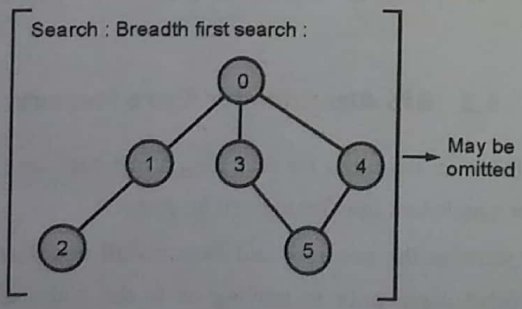
☞ **Algorithm**

(1) Mark any node as starter or initial

(2) Explore and traverse un-visited nodes adjacent to starting node.

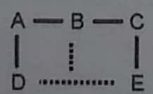
(3) Mark node as completed and move to next adjacent and un-visited nodes.

☞ **Search : Breadth first search**



BFS will always find the shortest path in an unweighted graph.

Consider an unweighted graph like this :



And my goal is to get from A to E.

I begin at A as it my origin. I queue A, followed by immediately dequeueing A and exploring it. This yields B and D, because A is connected to B and D. I thus queue both B and D.

I cheque B and explore it, and find that it leads to A (already explored), and C, so I queue C. I then dequeue D, and find that it leads to E, and that is my goal.

I then deque C, and find that it also leads to E, my goal.

- BFS can only be used to find shortest path in a graph if
  1. There are no loops.
  2. All edges have same weight or no weight.
- To find the shortest path, all you have to do is start from the source and perform a breadth first source and stop when you find your destination node.
- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantage of both algorithms.

3.6.4 Execution of BFS Algorithm

- **Step 1 :** Start by putting any one of the graph's vertices at the back of the queue.
- **Step 2 :** Take the front item of the queue and add it to the visited list.
- **Step 3 :** Create a list of that vertex's adjacent nodes.
- **Step 4 :** Keep continuing steps two and three till the querry is empty.

3.6.5 Advantages and Disadvantages of BFS

☞ **Advantages**

(1) Solution will be definitely found out by BFS if there is some solution.

(2) BFS will never get trapped in blind valley, means unwanted nodes.

(3) If there are more than one solution then it will find solution with minimal steps.

Disadvantages

(1) Memory constraints as it stores all the nodes of present level to go for next level.

(2) If solution is far away then it consumes time.

3.6.6 Applications of BFS Algorithm

We mention some of the applications where a BFS algorithm implementation can be highly effective.

(1) **Unweighted graphs :** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.

(2) **P2P Networks :** BFS can be implemented to locate all the nearest or neighbouring nodes in a peer to peer network. This will find the required data faster.

(3) **Web Crawlers :** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.

(4) **Navigation Systems :** BFS can help find all the neighbouring locations from the main or source location.

(5) **Network Broadcasting :** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

- It is possible to run BFS recursively without any data structures, but with higher complexity.
- DFS, as opposite to BFS, uses a stack instead of a queue, so it can be implemented recursively. Note that the code used is iterative but it is trivial to make it recursive.
- In BFS, a queue data structure is used. One can mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.

3.6.7 Performance Measures of BFS

Time Complexity

Breadth-first search, being a brute search generates all the nodes for identifying the goal. The amount of time taken for generating these nodes is proportional to the depth d and branching factor b and is given by,

$$1 + b + b^2 + b^3 + \ldots + b^d \simeq b^d$$

Hence the time-complexity = $O(b^d)$

Space Complexity

Unlike depth-first search wherein the search procedure has to remember only the paths it has generated, breadth-first search procedure has to remember every node it has generated. Since the procedure has to keep track of all the children it has generated, the space-complexity is also a function of the depth d and **branching factor b**. Thus space complexity becomes,

$$1 + b + b^2 + b^3 + \ldots + b^d \approx b^d$$

Hence the space-complexity = O (bd)

3.6.8 Limitations of Breadth First Search

1. Amount of time needed to generate all the nodes is considerable because of the time-complexity.
2. Memory constraint is also a major hurdle because of the space-complexity.
3. The searching process remembers all unwanted nodes which is of no practical use for the search.

GQ. Which storage structure is preferably chosen for node representation in open list, while performing best-first search over a state space and why?

OPEN is a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.

3.6.9 Example of Breadth First Search

GQ. Give an example of a problem for which breadth-first would work better than depth first search and vice-vesa.

In general, BFS is better for problems related to finding the **shortest paths** or somewhat related problems. Because here we can go from one node to all node that are adjacent to it and hence we effectively move from path length one to path length two and so on.

- While DFS on the other hand, helps more in connectivity problems and also in finding cycles in graph (cycles can be found in BFS with a bit of modification). Determining connectivity with DFS is trivial, if we call the explore procedure twice from the DFS procedure, then the graph is disconnected (this is for an undirected graph). We can see the strongly connected component algorithm for a directed graph here, which is a modification of DFS. Another application of the DFS is topological sorting.

Module 3

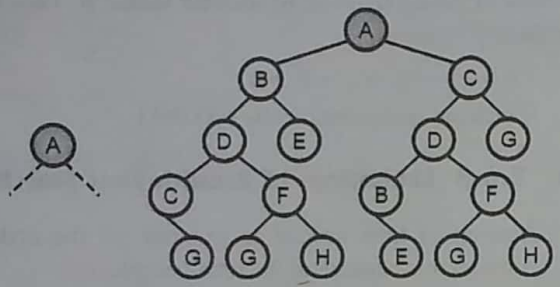
3.6.10 Solved Example on BFS

UEx. 3.6.2 (MU - Q. 5(A), Dec. 17, 10 Marks)

Consider the given tree, apply breadth first search algorithm and also write the order in which 10 nodes are expanded.

Soln. :

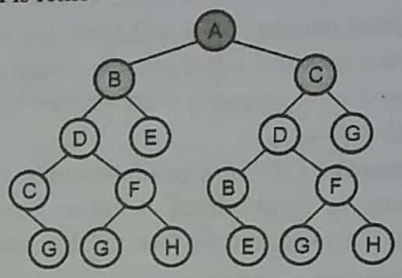
Step 1 :
Node, A can be regarded as source node.



(1B1)**Fig. Ex. 3.6.2**

FRINGE : A

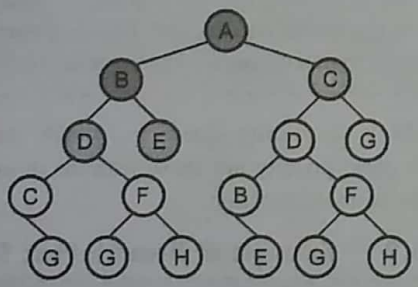
Step 2 : We consider sub-nodes B and C of A. Source node A is removed.



(1B2)**Fig. Ex. 3.6.2(a)**

FRINGE : B C

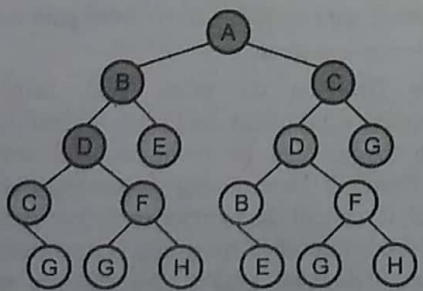
Step 3 : We begin with left nodes. Consider left node B and its subnodes D and E. We remove node B..



(1B3)**Fig. Ex. 3.6.2(b)**

FRINGE : D E D G

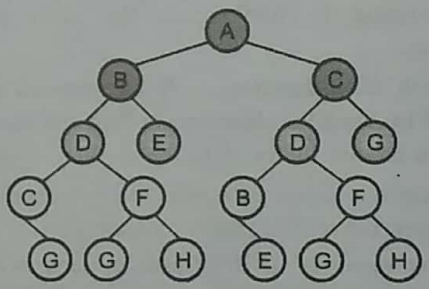
Step 5 : We consider left subnode C and its subnodes are D and G. We remove subnode C from the fringe and add subnodes D and C to the fringe.



(1B5)**Fig. Ex. 3.6.2(d)**

FRINGE : C D E

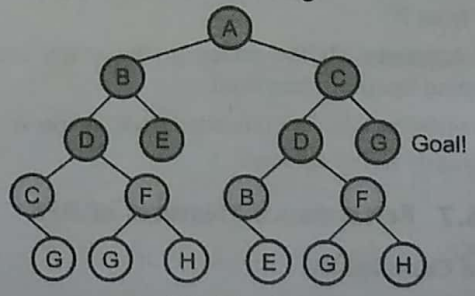
Step 4 : We consider left subnode D. Its subnodes are C and F. We remove node D.



(1B4)**Fig. Ex. 3.6.2(c)**

FRINGE : E D G C F

Step 6 : Now, consider right subnode E of B and subnode E is removed from fringe.



(1B6)**Fig. Ex. 3.6.2(e)**

Step 7 : Now, subnode D is expanded.

∴ Fringe : G C F B F

Step 8 : Now, we select G for expansion. It is found to be a goal node. And the algorithm terminates.

☞ **Performance Evaluation**

- **Time complexity :** $O(b^d)$
- **Space complexity :** O(bd)

  Where b - branching factor, and d - Depth of the shallowest goal node
- **Optimality :** can be reached

☞ **Advantage**

Path of minimal length to the goal can be found out.

☞ **Disadvantages**

1. The search-space should be small in order to use Breadth First Search (BFS) algorithm.
2. Generation and storage of a tree is required whose size is exponential (b^d), the depth of the shallowest goal node.

3.6.11 Application of BFS

1. Finding shortest path.
2. Checking graph with bipertiteness.
3. Copying Cheney's algorithm.

3.7 UNIFORM COST SEARCH

- Uniform-cost-search is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination.
- Nodes are expanded, starting from the root, according to the minimum cumulative cost. The uniform-cost search is then implemented using a Priority Queue.
- Here, instead of inserting all vertices into a priority queue, we insert only source, then one by one we insert nodes when needed.

3.7.1 Algorithm of U.C.S.

- Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost.
- This search is an uninformed search algorithm, i.e. it does not take the state of the node or search space into consideration.
- It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

☞ **Algorithm of Uniform Cost Search : (in AI)**

- **Step 1 :** Insert Root Node into the queue.
- **Step 3 :** Repeat till queue is not empty.
- **Step 3 :** Remove the next element with the highest priority, from the queue.
- **Step 4 :** If the node is a destination node, then print the cost and the path and exit, else insert all the children of removed elements into the queue with their cumulative cost as their priorities.

Here root Node is the starting node for the path, and a priority queue is being maintained to maintain the path with the least cost to be chosen for the next traversal.

3.7.2 Execution of Algorithm of Uniform Cost Search

- Uniform-cost search is similar to Dijkstra's algorithm. In this algorithm,

- **Step 1 :** from the starting state we will visit the adjacent states and will choose the least costly state.
- **Step 2 :** then we choose the next costly state from the all un-visited and adjacent states of the visited states,
- **Step 3 :** in this way we try to reach the goal state.

Module 3

Remark

Even if we reach the goal state we continue searching for other possible paths (if there are multiple goals).

- The elements in the priority queue have almost the same costs at a given time, and thus the name Uniform Cost Search.
- It may appear that elements do not have almost the same costs, but when applied on a much larger graph it is certainly so.
- Uniform costing refers to acceptance of identical costing principles and procedures by all or many units in the same industry by mutual agreement.
- 'Uniform Cost Search (UCS)' algorithm is mainly used when the step costs are not the same but we need the optimal solution to the goal state. In such cases, we use Uniform Cost Search, to find the goal and the path including the cumulative cost to expand each node from the root node to the goal node.
- Uniform cost-search is optimal. This is because, at every step the path with the least cost is chosen, and paths never get shorter as nodes are added, ensuring that the search expands nodes in the order of their optimal path cost. To measure the time complexity, we need the help of path cost instead of depth d.

3.7.3 Example of Uniform Cost Search (Linear Displacement)

Consider the example if Fig. 3.7.1; where we need to reach any one of the destination node $\{G_1, G_2, G_3\}$ starting from node S.

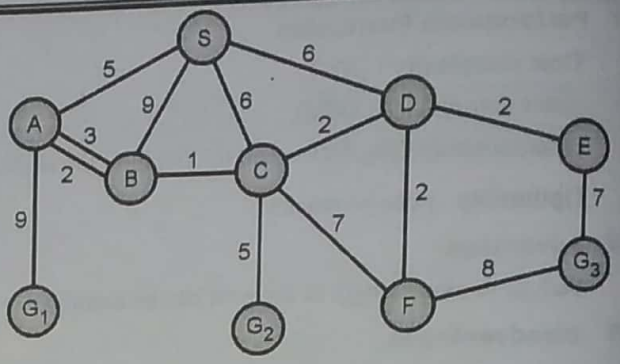


Fig. 3.7.1

Node { A, B, C, D, E and F } are the intermediate nodes. Our motive is to find the path from S to any of the destination state with the least cumulative cost. Each directed edge represents the direction of movement allowed through that path, and its labelling represents the cost is one travels through that path.

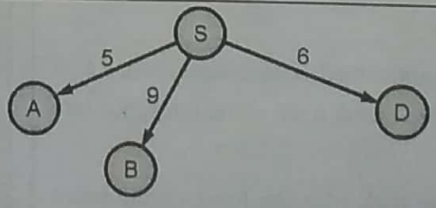
Thus overall cost of the path is a sum of all the paths.

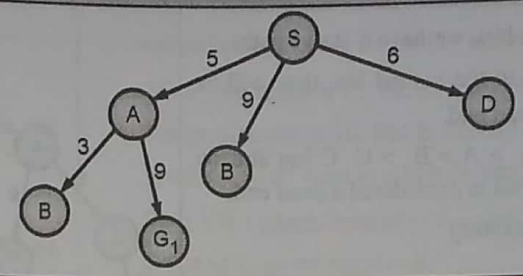
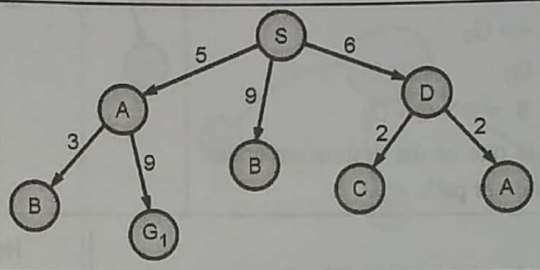
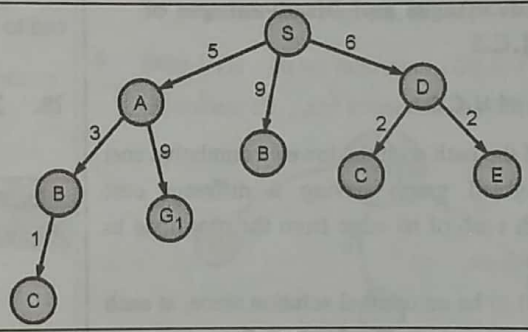
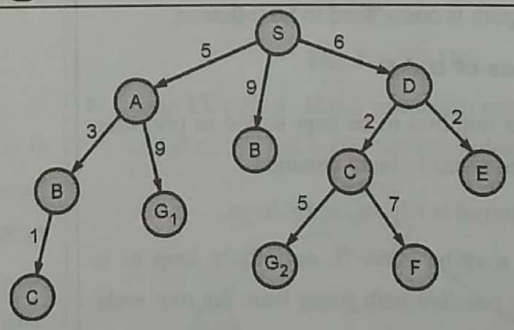
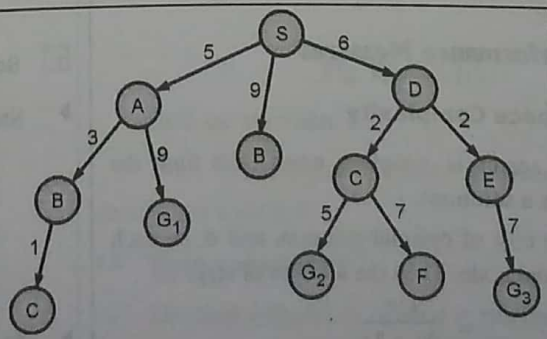
For e.g. : A path from S to G_1 –

{ S > A > G_1 } whose cost is $SA + AG_1 = 5 + 9 = 14$.

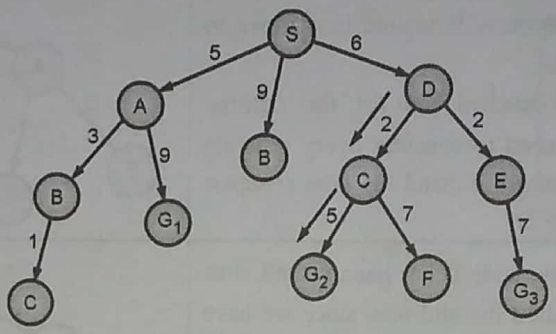
Here we maintain a priority queue the same as BFS with the cost of the path as its priority, lower the cost higher is the priority.

We use a tree to show all the paths possible and also maintain a visited list to keep track of all the visited nodes as we need not visit any node twice.

Explanation	Flow	Visited List
▶ **Step I :** We start with start node and check if we have reached any of the destination, nodes, i.e. No thus continue.		
▶ **Step 2 :** We reach all the nodes that can be reached from S i.e. A, B, D. And since node S has been visited, thus added to the visited list. Now we select the cheapest path first for further expansion i.e. A.		S

Explanation	Flow	Visited List
Step 3 : Node B and G_1 can be reached from A and since node A is visited thus move to the visited list. Since G_1 is reached but for the optimal solution, we need to consider every possible case, thus, we will expand the next cheapest path i.e. S –> D		S, A
Step 4 : Now, node D has been visited, thus it goes to visited list and now since we have three paths with the same cost, we choose alphabetically thus will expand node B.		S, A, D
Step 5 : From B, we can only reach node C. Now the path with minimum weight is S –> D –> C i.e. 8 Thus expand C, and B has now visited D node.		S, A, D, B
Step 6 : From C we can reach G_2 and F node with 5 and 7 weights respectively. Since S is present in the visited list, thus we are not considering the C –> S path. Now, C will enter the visited list. Now the next node with the minimum total path is S > D > E , i.e. 8 Thus we will expand E.		S, A, D, B, C
Step 7 : From E we can reach only G_3. E will move to the visited first.		S, A, D, B, C, E

Module 3

Explanation	Flow	Visited List
▶ **Step 8 :** In the last, we have 6 active paths. S – > B B is in the visited list, thus will be marked as a dead end. Same for S > A > B > C C has already been visited thus is considered a dead end. Out of the remaining $S > A -> G_1$ $S > D -> C -> G_2$ $S > D > E > G_3$ Minimum is $S > D > C > G_2$ and also G_2 is one of the destination nodes. Thus we found our path.		S, A, D, B, C, E

3.7.4 Advantages and Disadvantages of U.C.S

Advantages of U.C.S

1. It helps to find the path with the lowest cumulative cost inside a weighted graph having a different cost associated with each of its edge from the root node to the destination node.
2. It is considered to be an optimal solution since, at each state, the least path is considered to be followed.

Disadvantages of *U.C.S*

1. The open list is required to be kept sorted as priorities in priority queue needs to be maintained.
2. The storage required is exponentially large.
3. The algorithm may be stuck in an infinite loop as it considers every possible path going from the root node to the destination node.

3.7.5 Performance Measures

Time and Space Complexity

Uniform cost search is complete, when UCS finds the solution, (if there is a solution).

Let C* be the cost of optimal solution, and ϵ be each step closer to the goal node. Then the number of steps is,

$$= \frac{C^*}{(\epsilon + 1)}$$

Here, we have taken + 1, as we start from state 0 and end to $\frac{C^*}{\epsilon}$.

3.7.6 Solved Example on Curved Displacement

UEx. 3.7.1 **MU - Q. 1(b), Dec. 16, 5 Marks**

Apply uniform cost search algorithm on given graph.

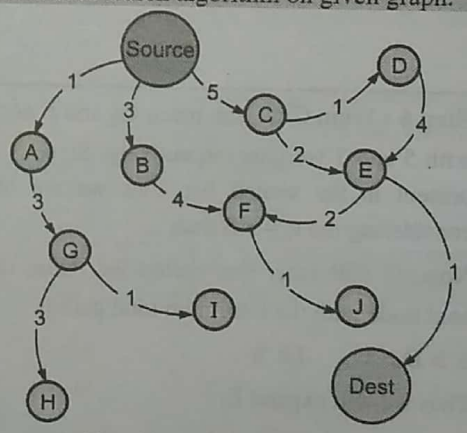


Fig. Ex. 3.7.1 : Graph

☑ **Soln. :**

▶ **Step I :** We mention source-node :

Fig. Ex. 3.7.1(a) : Node

▶ **Step II :** We add the nodes A, B, C to the source node, :

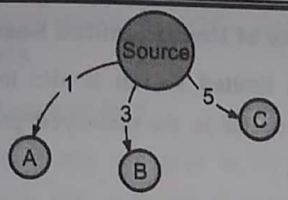


Fig. Ex. 3.7.1(b)

- **Step III :** The node A has minimum distance 1, so we keep it aside and add the node G.

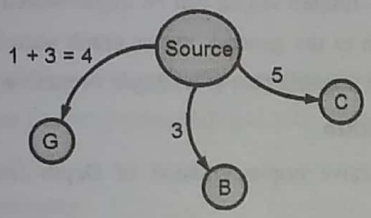


Fig. Ex. 3.7.1(c)

- **Step IV :** For nodes, B and C, B has minimum distance, so we add node F to B.

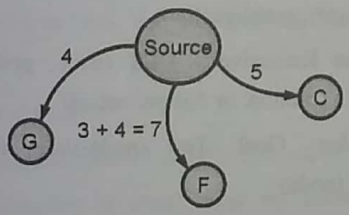


Fig. Ex. 3.7.1(d)

- **Step V :** Now, G has minimum distance, so we keep G aside and add H, note that C and I have same distance ; we remove C or I alphabetically ; but I has no further subnodes.

  Now, we remove D, and bring subnode E, with total distance of 10. But E has already lesser distance 7, so we add E with distance 7.

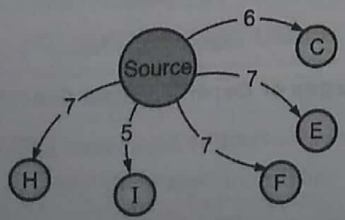


Fig. Ex. 3.7.1(e)

- **Step VI :** I has minimum distance, but I has no subnode, i.e. there is no further updating.

  Node, we remove D, but D has only one sub-node E, but its distance is 10. But E already exists with a lesser distance, so no need to add it further.

- **Step VII :** (Alphabetically) the next minimum distance is that of E, so we remove E.

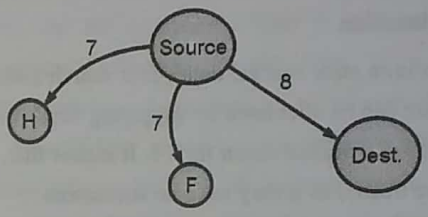


Fig. Ex. 3.7.1(f)

- **Step VIII :** Now, minimum cost is F, so it is removed (alphabetically) and subnode J is added.

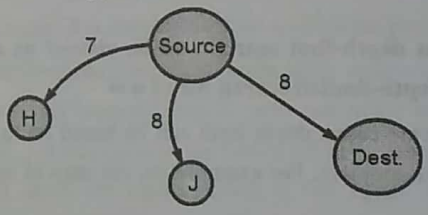


Fig. Ex. 3.7.1(g)

- **Step IX :** Now, H has minimum cost, so it is removed and H also H has no further subnodes.

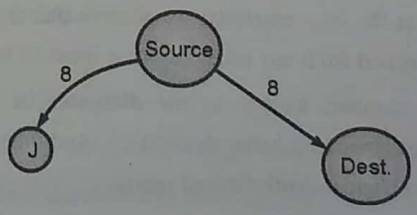


Fig. Ex. 3.7.1(h)

Here the algorithm ends.

Thus, the minimum distance between the source and destination node is **8**.

☞ **Time complexity**

The time complexity needed to run uniform cost search is : $O(b(1 + C / \varepsilon))$

Module 3

Where : b - branching factor, C - optimal cost, ε - cost of each step

Optimal :

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

3.8 DEPTH LIMITED SEARCH

(I) Explanation

In infinite state spaces, depth-first search method fails. This failure can be alleviated by supplying depth-first search with a pre -determined depth limit *l*. It means that, nodes at depth *l* are treated as if they have no successors.

This approach is called **depth-limited search**.

The depth limit can solve infinite path problem.

Its time-complexity is O (b^l) and its space-complexity is O (b*l*)

Thus depth-first search can be viewed as a special case of depth-limited search with $l = \infty$

In some cases, depth limit can be based on knowledge of the problem. For example, on the map of surat there are

25 cities.

So, if there is a solution, it must be of length 24 at the longest, hence $l = 24$ is a possible choice. But if we look at the map carefully, we observe that any city can be reached from any other city in at most 15 steps.

This number, known as the **diameter** of the **state space**, gives us a better depth limit. And this leads to a more efficient depth-limited search.

Observe that **depth-limited search** can terminate with two kinds of failure :

(i) the standard **failure value** which indicates no solution;

(ii) the **cut-off value** which indicates no solution within the **depth-limit**

☞ Drawbacks of Depth-Limited Search

(i) The **depth limited search** is also incomplete if we choose $l < d$; that is, the shallowest goal is beyond the depth limit.

(ii) Depth-limit search will also be non-optimal if we choose $l > d$.

Implementation

Depth –limited search can be implemented as a simple modification to the general tree-or graph search algorithm. Or, it can be implemented as a **simple recursive algorithm.**

(II) Algorithm

A recursive implementation of Depth-limited Search Algorithm

Function : Depth-Limited-search (Problem, limit) **returns** a solution, or failure/cut off.

Return : Recursive - DLS (Make-Node (Problem. Initial-state), problem, limit)

Function Recursive - DLS (node, problem, limit) **returns** a solution, or failure/ cut-off

If problem. Goal -Test (node-state) **then return** solution (node)

else if limit = 0 then **return cut off**

else

cut off-occurred ? ← false

for each action **in** problem. Action (node, state) **do**

child ← child – Node (problem, node, action)

result ← Recursive – DLS (child, problem, limit- 1)

If result = cutoff **then** cutoff-occurred? ← true

else if result ≠ failure **then return** result

If cutoff-occurred ? **then return** cutoff

else return failure

☞ Advantages of Depth Limited Search

1. Depth limited search is better than DFS as it requires less time and memory space.
2. DFS assures that the solution will be found if it exists in infinite time.

3. DLS has applications in graph theory particularly similar to DFS.

☞ Disadvantages of DLS

(i) The goal node may not exist in the depth limit set earlier, which will push the user to iterate further adding execution time.

(ii) The goal node cannot be found out if it does not exist in the desired limit.

Optimality

The DLS is a non-optimal algorithm since the depth that is chosen can be greater than d ($l > d$). Thus DLS is not optimal if $l > d$.

Time complexity

It is similar to DFS, i.e. O (b^l), where 1 is the specified depth limit.

Space complexity

It is similar to DFS, it is O (b^l), where 1 is the specified depth limit.

Conclusion – DLS

(i) DLS is not the case for uninformed search strategy.

(ii) DLS algorithm is used when we know the search domain, and there exists a prior knowledge of the problem and its domain.

(iii) There is little idea of the goal nodes depth.

(iv) The problem with depth-limited search is to set the value of l optimally, so as not to leave out any solution.

Also keep the time and space complexity to a minimum.

⏭ 3.9 ITERATIVE DEEPENING SEARCH TECHNIQUE (IDS OR IDDFS)

UQ. Explain Iterative Deepening search algorithms based on performance measure with justification; complete, optimal, Time and Space complexity.

(MU - Q. 4(a), Dec. 18, 10 Marks)

1. Iterative deepening search or more specifically iterative deepening depth-first search (IDS or IDDFS) is a **state space/graph search strategy** in which a **depth-limited version** of depth-first search is run repeatedly with increasing depth limits until the goal is found.
2. IDDFS is equivalent to breadth-first search, but uses much less memory; on each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth- first.
3. DDFS combines depth-first search's space-efficiency and breadth-first search's completeness (When the branching factor is finite). It is optimal when the **path cost is a non-decreasing function of the depth of the node**.
4. The **time complexity of IDDFS is O (b^d)** and its **space complexity is O(b^d)**, where **b** is the branching factor and **d** is the depth of the shallowest goal.
5. Since iterative deepening, visits states multiple times, it may seem wasteful, but it turns out to be not costly, since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times.

Module 3

3.9.1 IDDFS Algorithm

```
Function iterative-Deepening-search (problem) returns a solution or failure
inputs : problem, a problem
for depth ← 0 to ∞ do
result ← Depth-Limited-Search (problem, depth)
if result ≠ cutoff then return result
```

The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits.

It terminates when a solution is found or if the depth-limit search returns **failure** meaning that no solution exists.

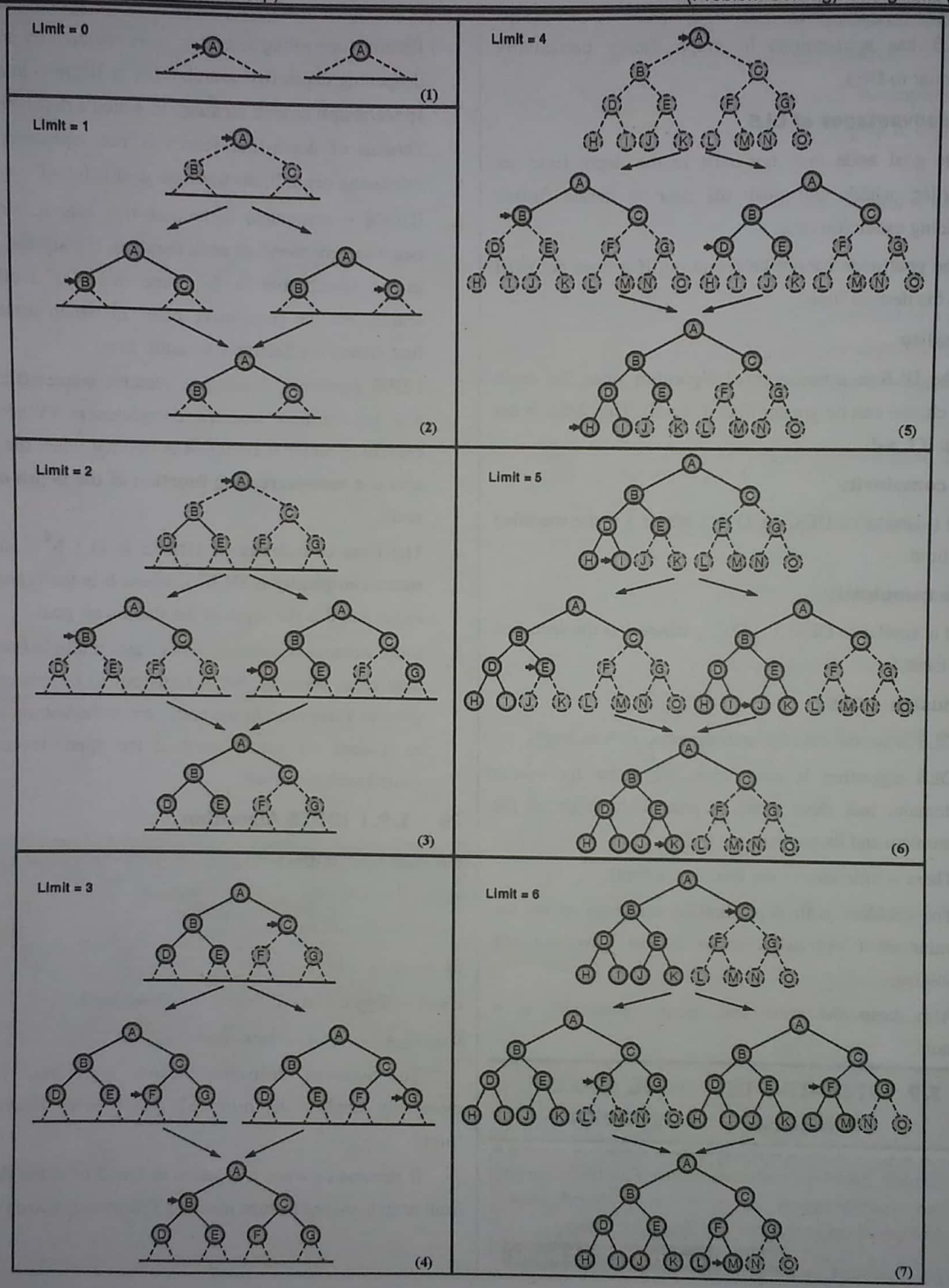


Fig. 3.9.1

3.9.2 Advantages of IDDFS

1. The **main advantage** of IDDFS in game tree searching is that the earlier searches tend to improve the commonly used heuristics, such as the killer heuristic and alpha- beta pruning, so that a more accurate estimate of the score of various nodes at the final depth search can occur and the search completes more quickly since it is done in a better order.

   For example, alpha-beta pruning is most efficient if it searches the best moves first.

2. A **second advantage** is the responsiveness of the algorithm. Because early iterations use small values for **d**, they execute extremely quickly. This allows the algorithm to supply early indications of the result almost immediately, followed by refinements as d increases, when used in an interactive setting. Such as in a chess-playing program, this facility allows the program to play at any time with the current best move found in the search it has completed so far. This can be phrased at each depth of the search core, producing a better approximation of the solution, though the work done at each step is recursive. This is not possible with a traditional depth-first search, which does not produce intermediate results.
3. The time complexity of IDDFS in well-balanced trees works out to be the same as **Depth-first search: O (b^d)**

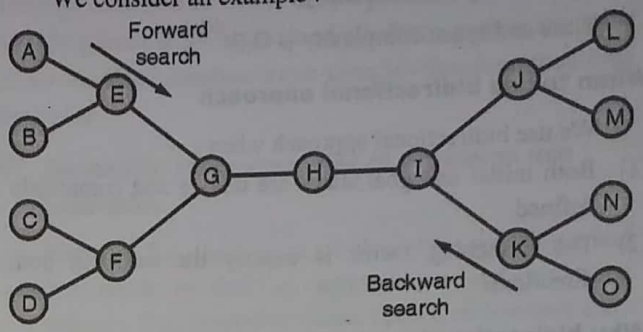
3.10 BIDIRECTIONAL SEARCH

The principle used in a bidirectional heuristic search algorithm is to find the shortest path from the current node to the goal node. The only difference being the two simultaneous searches from the initial point and from goal vertex. The main idea behind bidirectional searches is to reduce the time taken for search drastically.

This takes place when both searches happen simultaneously from the 'initial node depth' or 'breadth-first' and 'backwards from goal nodes'. They intersect somewhere in between of the graph.

The path traverses from the initial node through the intersecting point to goal vertex and that is the shortest path found because of this search.

We consider an example :



Bidirectional search algorithm

- **Step I :** Let A be initial node and O the goal node and H is the intersection node.
- **Step 2 :** We start searching simultaneously from start to goal node and backwards from goal node to start node.
- **Step 3 :** When the forward search and backward search intersect at one node, then searching stops.

Also observe that bidirectional searches are complete if a breadth-first search is used for both traversals, i.e., for both paths from start node till intersection and from goal node till intersection.

Two main types of bidirectional searches are as follows :

(1) Front to Back or B F E A

(2) Front to Front or B F F A

(1) Front to Back or B F E A

In bidirectional front, to front search, two heuristic functions are needed.

First is the estimated distance from a node to goal state using forward search and second, node to start state using reverse action. Here, h is calculated in the algorithm, and it is the heuristic value of the distance between the node n to the root of the opposite tree s or t. This is the most widely used bidirectional search algorithm.

(2) Front to front BFFA

Here the distance of all nodes is calculated, and h is calculated as the minimum of all heuristic distances from the current node to nodes on opposing fronts.

Performance measure

(1) **Completeness :** Bidirectional search is complete if BFS (Breadth-first search) is used in both searches.

(2) **Optimality :** It is optimal if BFS is used for search and paths have uniform cost.

Module 3

(3) Time and space complexity

Time and space complexity is O ($b^{d/2}$).

When to use bidirectional approach

We use bidirectional approach when :

(1) Both initial and goal states are unique and completely defined.

(2) The branching factor is exactly the same in both directions.

Why bidirectional approach ?

(i) In many cases it is faster, and it reduces the amount of required exploration.

(ii) Suppose if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity is O (b^d). But for two search complexity is O ($b^{d/2}$) which is far less than O (b^d).

3.11 INFORMED SEARCH

GQ. What is informed search ?

Informed search (heuristic search) : This can decide whether one non-goal state is more promising than another non-goal state. The advantages of the informed search comes from the fact that :

1. It adds **domain-specific information** to select the best path along which to continue searching.
2. Define a heuristic function **h(n)** that estimates the "goodness" of a node n. Specifically, **h(n) = estimated cost** (or distance) of minimal cost path from n **to a goal state.**
3. The heuristic function is an estimate of how close we are to a goal, based on domain-specific information that is computable from the current state description. Some of the examples of informed search are best first search, beam search, **A*** and **AO*** algorithms etc.

Informed search algorithms

Informed search algorithm contains an array of knowledge that tells us how far we are from the goal, path cost, how to reach to goal node etc. This knowledge helps agents to explore less to the search space and find the goal node more efficiently.

The informed search algorithm is more useful for large search space. Informed search uses the idea of heuristic, hence it is also called as Heuristic search.

3.11.1 Example for Informed Search

GQ. Give an example for informed search.

Beam search is an example for informed search

Beam search: This is an attractive heuristic search technique because it permits searching to be done on a **multi-processor machine**, thereby reducing computations. Reduction in computations is achieved by pursuing some paths and selecting only selected paths.

The searching process is similar to breadth-first search wherein searching proceeds level by level. At each level, heuristic functions are applied to reduce the number of paths to be explored. In fact, it is done to keep the width of the beam to be minimal. The width of the beam is fixed and whatever be the depth of the tree, the number of alternatives to be scanned is the product of the width and the depth.

3.11.2 Algorithm for Beam Search

- **Step 1 :** Let width_of_beam = w.
- **Step 2 :** Put the initial node on a list START.
- **Step 3 :** If (START is empty) or (START = GOAL), then terminate search.
- **Step 4 :** Remove the first node from START, Call this as node a.
- **Step 5 :** If (a = GOAL), then terminate search with success.
- **Step 6 :** Else if node a has successors, generate all of them and add them at the tail of START.
- **Step 7 :** Use a heuristic function to rank and sort all the elements of START.
- **Step 8 :** Determine the nodes to be expanded. The number of nodes should not be greater than w. Name these as START1.
- **Step 9 :** Replace START with START1.
- **Step 10 :** Goto Step 2.

Fig. 3.11.1 shows how beam search proceeds. This search has been used in an expert system called **ISIS for factory scheduling**.

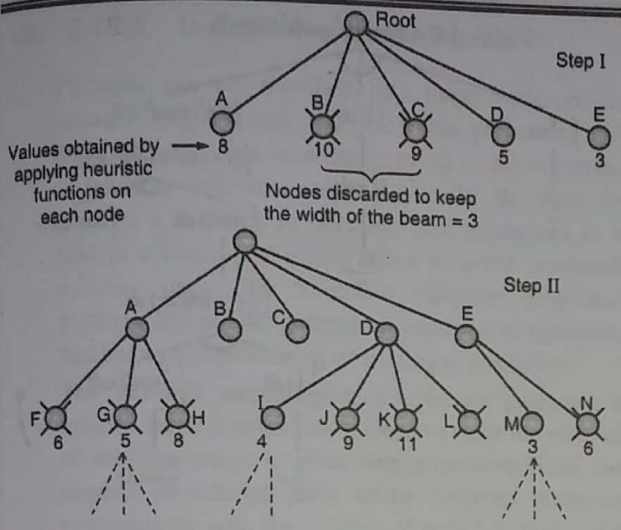


Fig. 3.11.1 : Beam search procedure

3.12 HEURISTIC FUNCTION

UQ. What is heuristic function ?

MU - Q. 3(b), Dec. 18, 10 Marks, Q. 1(c), May 18, 4 Marks, Q. 1(c), May 17, 4 Marks, Q. 3(b), Dec. 16, 5 Marks, Q. 1(c), Dec. 15, 3 Marks

The process of searching can be drastically reduced by the use of heuristics. Heuristics are approximations used to minimize the searching process.

- It is a function that maps from problem state description to measures of desirability, usually represented as number. Which aspect of the problem state are considered, how these aspects are evaluated, and weight given to the individual aspects are chosen. Define a heuristic function h(n) that estimates the "goodness" of a node n.

  Specifically, **h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.**

- The heuristic function is an estimate of how close we are to a goal, based on domain-specific information that is computable from the current state description.

Computed in such a way that the value of the heuristic function at a given node in the search give as good estimate as possible of whether that node is on the desired path to a solution.

Well-designed heuristic function can play an important role in efficiently guiding a search process toward a solution. Sometimes very simple heuristic function can provide a fairly good estimate of whether a path is any good or not. In other situation more complex function should be employed.

Generally, two categories of problems use heuristics

1. Problems for which no exact algorithms are known and one needs to find an approximate and satisfying solution. E.g., computer vision, speech recognition etc.
2. Problems for which exact solutions are known, but computationally infeasible. E.g., Rubik's cube, chess etc. The heuristics which are needed for solving problems are generally represented as a heuristic function which maps the problem states into number. These numbers are then appropriately used to guide search.

3.12.1 Simple Heuristic Functions

Module 3

1. In the famous 8-tile puzzle, the hamming distance is a popular heuristic function. It is an indicator of the number of tiles in the position they are to be.
2. In a game like chess, the material advantage one has over the opponent is an indicator. Normally, the following values are assigned to the pieces. Queen-9, etc.

The following algorithms make use of heuristic evaluation functions

1. Hill climbing
2. Constraint satisfaction
3. Best-first search
4. A* algorithm
5. AO* algorithm
6. Beam search

- The purpose of heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available.
- The more accurate the heuristic function estimate the true merit of each node in the search tree, the more direct the solution process.
- In the extreme, the heuristic would be so good that essentially no search would be required system would move directly to a solution.
- But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process.

- In general, there is a trade-off between the cost of evaluating a heuristic function and the saving of search time that the functions provide.

3.12.2 Problem Characteristics for Heuristic Search

GQ. What are the various problem characteristics for heuristic search? **Or** Explain the problem characteristic briefly with appropriate example.

Heuristic search is a very general method applicable to a large class of problems. It encompasses a variety of specific techniques, each of which is particularly effective for a small class of problems.

Several key dimensions for Heuristic Search

1. Is the problem decomposable into a set of (nearly) independent smaller or easier sub problems?
2. Can solution steps be ignored or at least undone if they prove unwise?
3. Is the problem's universe predictable?
4. Is a good solution to the problem obvious without comparison to all other possible solutions?
5. Is the desired solution a state of the world or a path to a state?
6. Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?
7. Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction between the computer and a person?

3.12.3 Is the Problem Decomposable ?

A very large and composite problem can be easily solved if it can be broken into smaller problems and recursion could be used. Suppose we want to solve.

Ex : $\int (x^2 + 3x + \sin^2 x \cos 2x)\, dx$

This can be done by breaking it into three smaller problems and solving each by applying specific rules. On adding the results, complete solution is obtained.

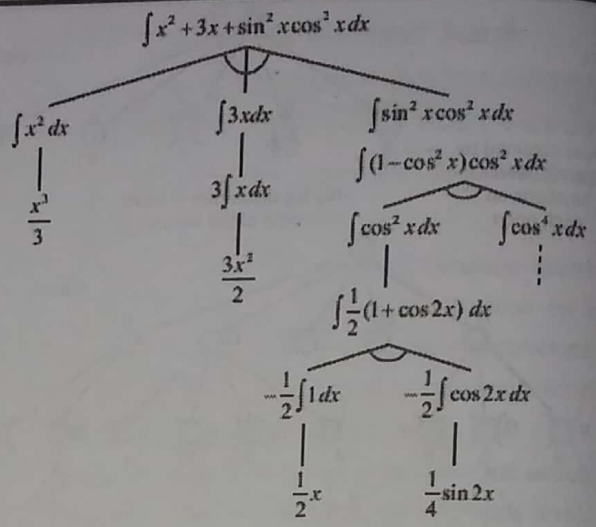


Fig. 3.12.1

3.12.4 Can Solution Steps be Ignored or Undone ?

- Problem fall under three classes ignorable, recoverable and irrecoverable. This classification is with reference to the steps of the solution to a problem. For example, consider proving the theorem. We may later find that it is of no help. We can still proceed further, since nothing is lost by this redundant step. This is an example of ignorable solutions steps.
- Now consider the 8 puzzle problem tray and arranged in specified order. While moving from the start state towards goal state, we may make some stupid move and consider theorem proving. We may proceed by first proving lemma. But we may backtrack and undo the unwanted move. This only involves additional steps and the solution steps are recoverable.
- Lastly consider the game of chess. If a wrong move is made, it can neither be ignored nor be recovered. The thing to do is to make the best use of current situation and proceed. This is an example of an irrecoverable solution steps.

(i) Ignorable problems

Ex : Theorem proving

In which solution steps can be ignored.

(ii) Recoverable problems Ex: 8 puzzle

In which solution steps can be undone.

(iii) Irrecoverable problems Ex : Chess

In which solution steps can't be undone.

A knowledge of these will help in determining the control structure.

3.12.5 Is the Universal Predictable ?

- Problems can be classified into those with certain outcome (eight puzzle and water jug problems) and those with uncertain outcome (playing cards) in certain – outcome problems, planning could be done to generate a sequence of operators that guarantees to a lead to a solution. Planning helps to avoid unwanted solution steps. For uncertain outcome problems, planning can at best generate a sequence of operators that has a good probability of leading to a solution.
- The uncertain outcome problems do not guarantee a solution and it is often very expensive since the number of solution and it is often very expensive since the number of solution paths to be explored increases exponentially with the number of points at which the outcome cannot be predicted. Thus one of the hardest types of problems to solve is the irrecoverable, uncertain-outcome problems. Ex : Playing cards.

3.12.6 Is Good Solution Absolute or Relative ? (Is the Solution a State or a Path?)

- There are two categories of problems. In one, like the **water jug and 8 puzzle problems**, we are satisfied with the solution, unmindful of the solution path taken, whereas in the other category not just any solution is acceptable.
- We want the best, like that of travelling salesman problem, where it is the shortest path. In any path problems, by heuristic methods we obtain a solution and we do not explore alternatives. For the best path problems all possible paths are explored using an exhaustive search until the best path is obtained.

3.12.7 The Knowledge Base Consistent ?

- In some problems the knowledge base is consistent and in some it is not. For example, consider the case when a Boolean expression is evaluated.
- The knowledge base now **contains theorems and laws of Boolean algebra which are always true**. On the contrary, consider a knowledge base that contains facts about production and cost.
- These keep varying with time. Hence many reasoning schemes that work well in consistent domains are not appropriate in inconsistent domains. **Ex:** Boolean expression evaluation.

Remark

- A Boolean expression is a logical statement that is either True or False.
- Boolean expressions can compare data of any type as long as both parts of the expression have the same basic data type.
- One can test data to see, if it is equal to, greater than, or less than other data.

Boolean data is as follows :

Boolean values :

(Yes and no, and their synonyms, on and off, and true and false).

3.12.8 What is the Role of Knowledge ?

- Though one could have unlimited computing power, the size of the knowledge base available for solving the problem does matter in arriving at a good solution. Take for example, the game of playing chess, just the rules for determining legal moves and some simple control mechanism is sufficient to arrive at a solution. But, additional knowledge about good strategy and tactics could help to constrain the search and speed up the execution of the program. The solution would then be realistic.
- Consider the case of predicting the political trend. This would require an enormous amount of knowledge even to be able to recognize a solution, leave alone the best. Ex : Playing chess, newspaper understanding.

3.12.9 Does the Task requires Interaction with the Person

The problems can again be categorized under two heads as follows :

(i) Solitary Problems
(ii) Conversational Problems

(i) Solitary Problems

In which the computer will be given a problem description and will produce an answer, with **no intermediate communication** and with the demand for an explanation of the reasoning process. Simple theorem proving falls under this category. Given the basic rules and laws, the theorem could be proved, if one exists. **Ex:** Theorem proving (give basic rules and laws to computer)

Module 3

(ii) Conversational Problems

In which there will be **intermediate communication between a person and the computer**, whether to provide additional assistance to the computer or to provide additional informed information to the user, or both problems such as medical diagnosis fall under this category, where people will be unwilling to accept the verdict of the program, if they cannot follow its reasoning. Ex : Problems such as medical diagnosis.

3.12.10 Problem Classification

Actual problems are examined from the point of view; the tasks here is examining an input and decide which of a set of known classes. **Ex :** Problems such as medical diagnosis, engineering design.

3.12.11 Heuristic Function for : Travelling Salesman Problem

GQ. Explain with suitable example, heuristic function for : travelling salesman problem

UQ Give the initial state, goal test, successor function, and cost function for the travelling salesperson problem (TSP). There is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city. **(MU - Q. 5(A), Dec. 16, 6 Marks)**

UQ. Define heuristic function. Give an example heuristics functions for 8-puzzle problem. Find the heuristics value for a particular state of the Blocks World Problem. **(MU - Q. 1(b), May 17, 5 Marks)**

Travelling salesman problem : A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

Why informed (Heuristic) search method?

Informed search method are more problem specific and acquires related knowledge and hence finds solution more efficiently than uniformed method. **Exploration :** Exploration is one of the technique used for information gathering.

There are a few possible issues with this algorithm. One of the most important issues is completeness. It can be a typical extension to generate and test using the knowledge that directs the search. This knowledge is based on heuristic function that detects the closeness of current state to the goal state.

Example : Let us assume that we are solving travelling salesman problem with hill climbing.

1. Begin with the initial state (that is a city salesman has visited).
2. Move in the direction so that the city is not repeated, but additional city is visited (better state).
3. Heuristic can be the number of cities visited.
4. Keep moving in the state that improves the number of cities with legal action.
5. When there is no way to improve heuristic function, stop.

Hill climbing algorithm makes incremental changes in the optimal direction based on heuristic.

There are many variants of hill climbing algorithms :

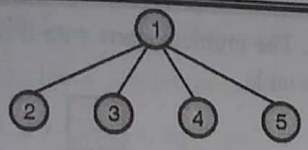
- To apply the steepest ascent hill climbing to the travelling salesman problem, we need to consider all successor states (possible moves) to choose the best one. This algorithm has higher complexity than the basic hill climbing, since it requires selecting move from a number of possible moves.
- Both basic and steepest ascent hill climbings may not able to reach the goal state, and hence, may fail to find the solution. Generally, it terminates in finding a state that does not have a better state in the vicinity or from where it is not possible to get to a better state in a single move.

3.12.12 Branch And Bound Technique

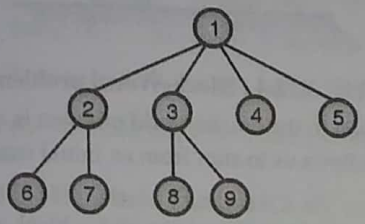
GQ. What do you mean by branch and bound technique? Give some examples.

Branch and bound is a state space search method in which all the children of a node are generated before expanding any of its children.

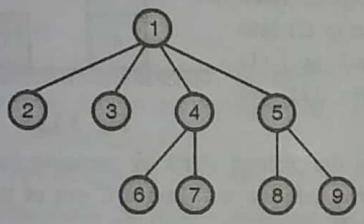
1. Live-node : A node that has not been expanded.
2. It is similar to backtracking technique but uses BFS-like search.
3. Dead-node : A node that has been expanded.



Live node : 2, 14.2 and 5



FIFO Branch and Bound Children of E-node are inserted in a queue
Fig. 3.12.2



LIFO Branch and Bound Children of E-node are inserted in a stack
Fig. 3.12.3

3.12.13 Block World Problem

UQ. Define heuristic function. Give an example heuristics function for Blocks world problem.
(MU - Q. 1(a), Dec. 2015, 5 Marks)

UQ. Find the heuristics value for a particular state of the Blocks World Problem.
(MU - Q. 1(b), May 17, 5 Marks)

- The Block World is a 'planning domain' in artificial intelligence. The algorithm is similar to a set of wooden blocks of various shapes and colours but of same dimensions, sitting on table.
- The goal is to build one or more vertical stacks of blocks.
- The blocks in a world are infinite. Though one can see only a chunk of the world at a time. The chunk area is 16 blocks width by 16 block length by 256 blocks height, meaning that one can only see a total volume of 65,536 blocks.
- Artificial intelligence blocks is an intuitive WYSIWYG (What you see is what you get) interface that allows users to create machine learning models.
- The idea of AI-blocks is to have a simple scene with draggable objects that have scripts attached to them.
- The Blocks World is one of the most famous planning domains in Artificial Intelligence.
- A solution to the blocks world problem is a sequence of actions that allows us to start from an initial state and end up to a goal state. Therefore the optimal solution is the solution with the minimum cost (or else minimum actions required to get from the initial state to the goal state).

3.12.14 Actions

- The first member of the tuple represents the block that is about to move and second member of the tuple represents the block (or table) that will have the block (first member of the tuple) stacked on top of it.

  For example : If possible action = [(1, 2), (3, 0)] that means we have two actions :

  (i) The first action will move block 1 on top of block 2, and

  (ii) Second action will move block 3 on the table. (One cannot move a block that has other blocks stacked on top of it).
- After that we have list of all valid actions (list of tuples) and we are ready to move on and apply those actions to a state.
- From an algorithm perspective, blocks-world is an np-hard search and planning problem.
- The task is to bring the system from an initial state into a goal-state.

  [**Remark** : 'NP-complete' is non-deterministic polynomial-time complete. In computational complexity theory, a problem for which the correctness of each solution can be verified quickly and a brute-force search algorithm can actually find a solution by trying all possible solutions].

3.12.15 Motivation

- The simplicity of this toy-world (block world) / ends itself to classical symbolic AI approaches, in which the world is modelled as a set of abstract symbols, which may be reasoned about).
- AI can be researched in theory and with practical applications.
- To develop program on AI system, we invent an easy to solve domain which is called a 'toy-problem'.
- Toy problems were invented with the aim to program

Module 3

an AI which can solve it.

- The blocks-world domain is an example of a toy-problem. Its major advantage over more realistic AI applications is that, many algorithms and software programs are available which can handle the situation. This allows to compare different theories against each-other.
- More complicated derivations of the problem consist of cubes in different sizes, shapes and colours. From an algorithm perspective, blocks world is an np-hard search and planning problem.
- The task is to bring the system from an initial state into a goal state.

3.12.16 Symbolic Representation

(A) Four actions

(1) Unstack (A, B) : pick up **clear** block A from block B.

(2) Stack (A, B) : place block A using the arm onto **clear** block B.

(3) pickup (A) : lift clear block A with the empty arm.

(4) putdown (A) : place the held block A onto a free space on the table.

(B) Five predicates

(i) On (A, B) : block A is on block B.

(ii) On table (A) : block A is on table.

(iii) Clear (A) : block A has nothing on it.

(iv) Holding (A) : the arm holds block A.

(v) Arm empty : the arm holds nothing.

Remark : We use the blocks world as an example, because :

(i) it is sufficiently simple and well-defined.

(ii) easily understood.

(iii) yet still provides a good sample environment to study planning.

(iv) problems can be broken into nearly distinct subproblems.

Here, we can show how partial solutions need to be combined to form a realistic complete solution.

3.12.17 Sussman Anomaly

This is a problem in Artificial Intelligence described by Gerald Sussman.

In the problem, three blocks (labelled A, B and C) rest on table. The agent must stack the blocks such that A is a top B, which in turn is a top C. But it may only move one block at a time. The problem starts with B on the table, C a top A and A on table.

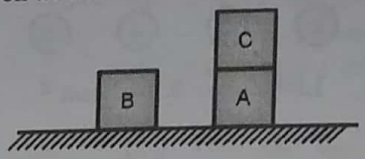


Fig. 3.12.4 : Block-World problem

A solution to the blocks world problem is a sequence of actions that allows us to start from an initial state and end up to a goal.

A mechanical **robot** arm can pick a block and place the cubes either on the table or on top of another block.

A planner typically separate the goal (stack A a top B atop C) into subgoals, such as : (1) get A a top B ; (2) get B a top C

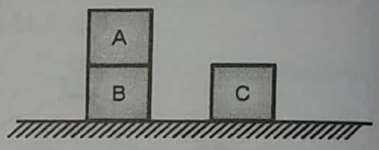


Fig. 3.12.5

Suppose the planner starts by pursuing subgoal 1, the straightforward solution is to move C out of the way, then move A a top B.

While sequence accomplishes subgoal (1), the agent cannot pursue subgoal (2) without undoing subgoal (1), since both A and B must be moved atop C.

Fig. 3.12.6

If instead the planner starts with Goal 2, i.e. to move B. But again the planner cannot pursue Goal 1 without undoing Goal 2.

Thus one of the solutions is

(i) unstack (C, A)

(ii) put down (C)

(iii) stack (B, C)

(iv) stack (A, B) and we get

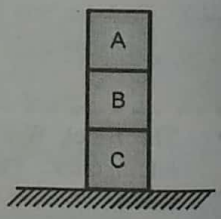


Fig. 3.12.7

Remark

Sussman believed that intelligence requires a list of **exceptions** or **tricks**, and developed a modular planning system for 'debugging' plans.

Most modern planning systems handle this anomaly, but it is still useful for explaining why '**planning** is **non-trivial**'.

First order logic

(i) On table (A) – block A is on the table

$$\neg \exists x \quad On(x, A) \quad or$$

$$\forall x \neg \quad On(x, A) \quad [i.e\ On(x, A) \rightarrow x \text{ is on A}].$$

(ii) Initially : [clear (B) $\wedge$ on (C, A) $\wedge$ on table (A)]

(iii) Solution : [unstack (C, A) $\wedge$ put down (C)]

$\wedge$ clear (C) $\wedge$ stack (B, C) $\wedge$ stack (A, B)]

3.12.18 Heuristic Function : H_1

- This function gives better results. It requires less explored nodes and less execution time.
  - (i) First we check how many blocks have different positions.
  - (ii) Compare the node-state and goal-state (basically how many blocks are not where they should be in the current node-state).
  - (iii) The H-function H_1 is admissible because it never overestimates the cost of reaching the goal and is also consistent because for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to ρ plus the estimated cost of reaching the goal from ρ.
- Thus heuristic function estimates the cost of getting from one place to another (from the current state to the goal state). Also called simply a heuristic.
- A heuristic function algorithm or simply heuristic is a shortcut of solving a problem when there are no exact solutions for it or the time to obtain the solution is too long.

Remark

Automated planning and scheduling problems are usually described in the planning Domain Definition Language (PDDL) notation which is an AI planning language for symbolic manipulation tasks. If something was formulated in PDDL notation, it is called a 'domain'. Therefore the task of stapling blocks is a block world domain, which stays in contrast to other planning problems like the 'dock worker robot' domain and the 'monkey and banana' problem.

3.12.19 Example of 'Block World Problem

(I) Local Heuristic

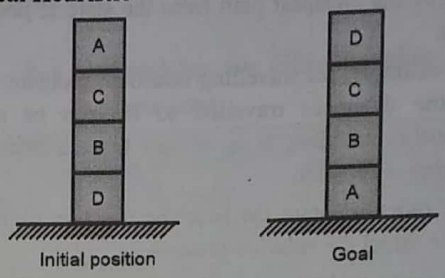


Fig. 3.12.8

(i) pickup (A) $\wedge$ pickup (C) $\wedge$ pickup (B) $\wedge$ pickup (D).

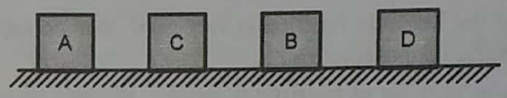


Fig. 3.12.9

(ii) Stack (B, A) $\wedge$ stack (C, B) $\wedge$ stack (D, C)

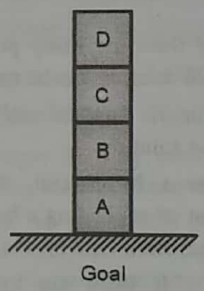


Fig. 3.12.10

(II) Global Heuristic

Here, every block has an incorrect support.

(i) pickup (A) $\wedge$ pickup (C) $\wedge$ pickup (B) $\wedge$ pickup (D).

(ii) stack (B, A)

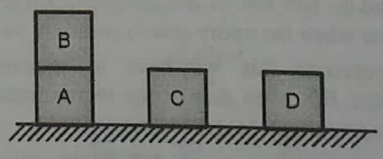


Fig. 3.12.11

(iii) stack (C, B) $\wedge$ stack (D, C)

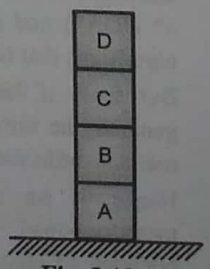


Fig. 3.12.12

Module 3

3.12.20 Properties of Heuristic Functions

(1) The heuristic function for a node n is, h (n) = estimated cost of the cheapest path from the state at node n to a goal.

For example, for travelling salesman problem, the sum of the distances travelled so far can be a simple heuristic function. It is of two types : Maximise or minimise function.

(2) The more accurate the heuristic function estimate, the more direct the solution process.

(3) For many problems, the cost of computing the value of such a function outweighs the effort saved in the search process.

(4) If the heuristic function is very good, then no search is required to a solution.

(5) The purpose of heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available.

(6) **Optimality :** If there are many possible solution to a problem, optimal solution can be found out.

(7) **Accuracy :** Heuristic function can yield approximately good and precise solution.

(8) **Execution time :** In general, there is a trade-off between the cost of evaluating a heuristic function and the saving of execution time for the function.

(9) **Completeness :** If there are many solutions to a problem, we have to find all of them and choose the appropriate one.

3.12.21 Memory Bounded Heurestic Search

- Generally, while carrying out the search out the search, we tend to run out of the memory before time. This happens when the entire search paths are stored.
- To overcome this we have to remember partial solutions. But even then space requirements, memory bounded heuristic searches are used.
- Recursive best first search (RBFS), iterative deepening a* (IDA*) and memory bounded A* (MA*) are the algorithms that fall under this search.
- But there is drawback at RBFS and IDA*. They generate the same states again and again. The cost of memory-utilisation is same for all the above methods.
- Hence, if we are having more memory, we use memory-bound variants MA* or simplified memory bounded A* algorithms, (SMA*).
- SMA* or simplified memory bounded A* is a shortest path algorithm based on the A* algorithm.
- The main advantage of SMA* is that it uses a bounded memory, while the A* algorithm might need exponential memory. All other characteristics of SMA* are inherited from A*

Properties of SMA*

(1) It works with heuristic, just as A*

(2) It is complete if the allowed memory is high enough to store the shallowest optimal solution.

(3) It is optimal if the allowed memory is high enough to store the shallowest optimal solution.

(4) It avoids repeated states as long as the memory bound allows it.

(5) It uses all memory available.

(6) Enlarging the memory bound of the algorithm will only speed up the calculation.

(7) When enough memory is available to contain the entire search tree, then calculation has on optimal speed.

Implementation

- The implantation of SMA* is very similar to that of A*; the only difference is that nodes with the highest f-cost are deleted from the queue when there is not any space left.
- Since these nodes are deleted, SMA* has to remember the f-cost of the best forgotten child of the parent node.
- When it appears that all explored paths are worse than such a forgotten path, the path is regenerated.

SMA* algorithm

- SMA* is a shortest path algorithm that is based on A* algorithm. The difference between SMA*a and A* is that SMA* uses a bounded memory, while the A* algorithm might need exponential memory.
- Like the A*, it expands the most promising branches according to the heuristic.
- SMA* just like A* evaluates nodes by combining g(n)m, the cost to reach the node, and h(n), the cost to get from the node to the goal : f(n) = g(n) + h(n).
- Since g(n) is the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.
- The lower the f value is, the higher priority the node will have. In addition, the node stores the f-value of the best forgotten successor.

3.13 LOCAL SEARCH ALGORITHMS

UQ. Write short note on : Local search Algorithms
(MU - Q. 6(b), May 18, Q. 6(c), May 19, 6 Marks)

- **In computer science local search** is a heuristic method for solving computationally hard **optimisation** problems.
- Local search starts from an initial solution and evolves the single solution that is mostly better solution.
- At each solution in this path it evaluates a number of moves on the solution and applies the most suitable move to take the step to the next solution. It continues this process for a high number of iterations until it is terminated.
- Local search uses a single search path and moves facts around to find a good feasible solution. Hence it is natural to implement.
- Local search algorithms are widely applied to numerous hard computational problems, including problems from **artificial intelligence, mathematics, operations research,** engineering and bioinformatics.

Some problems where local search is applied are :

(1) The **vertex cover problem,** in which a solution is a **vertex cover** of a **graph,** and the target is to find a solution with a minimal number of nodes.

(2) The **travelling salesman problem,** in which a solution is a **cycle containing** all nodes of the graph and the target is to minimise the total length of the cycle.

(3) The Hopfield Neural Networks problem for which finding stable configuration in Hop-field network.

3.14 HILL CLIMBING ALGORITHM

GQ. Briefly define hill climbing algorithm.

UQ. Explain Hill Climbing and its Drawback in details.
(MU - Q. 2(b), May 19, 5 Marks, Q. 6(b), May 18, 5 Marks, Q. 2(A), Dec. 17, 10 Marks, Q. 2(b), May 17, 10 Marks)

UQ. Explain Hill-climbing algorithm with an example.
(MU - Q. 1(d), Dec. 17, 5 Marks)

❑ **Definition :** This algorithm, also called **discrete optimization algorithm,** uses a simple heuristic function viz., the amount of distance the node is from the goal. The ordering of choices is a heuristic measure of the remaining distance one has to traverse to reach the goal node.

In fact, there is practically no difference between hill-climbing and depth-first search except that the children of the node that has been expanded are sorted by the remaining distance.

3.14.1 Algorithm for Hill-Climbing Procedure

- **Step 1 :** Put the initial node on a list START.
- **Step 2 :** If (START is empty) or (START = GOAL), then terminate search.
- **Step 3 :** Remove the first node from START. Call this as node a.
- **Step 4 :** If (a = GOAL), then terminate search with success.
- **Step 5 :** Else if node a has successors, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from the goal and add them at the beginning of START.
- **Step 6 :** Goto Step 2.

3.14.2 Explanation of Hill Climbing Algorithm

Hill-climbing technique is being used in some activity or other in our day-to-day routine. **Some of them are :**

1. While listening to somebody playing flute on the transistor, tone and volume control are adjusted in a way that makes the music melodious.
2. While tuning the carburettor of a scooter, the accelerator is raised to its maximum once and the carburetor is tuned so that the engine keeps on running for a considerably long period of time.
3. An electronics expert, while making the transistor for the first time, tunes the **radio set at mid-afternoon** when the signal is weak for proper reception.

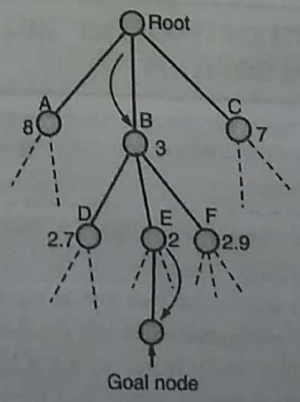


Fig. 3.14.1: Search tree for hill-climbing procedure

Module 3

3.14.3 Simple Hill Climbing Algorithm

GQ. Explain simple hill climbing algorithm with its limitations.

The simplest way to implement hill climbing is as follows :

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:

(i) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(ii) Evaluate the new state.

(a) If it is a goal state, then return it and quit.

(b) If it is not a goal state but it is better than the current state, then make it the current state.

(c) If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and generate-and-test is the use of an evaluation function as a way to **inject task-specific knowledge into the control process** makes this method heuristic search methods, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, we have asked the relatively vague question, **"Is one state better than another"** For the algorithm to work, a precise definition of better must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

3.15 STEEPEST-ASCENT HILL CLIMBING ALGORITHM

GQ. Explain steepest-ascent hill climbing algorithm with its limitations.

Steepest-ascent hill climbing : A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called **steepest-ascent hill climbing** or **gradient search**. Notice that this contrast with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows :

3.15.1 Algorithm : Steepest-ascent Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit.

   Otherwise, continue with the initial state as the current state.

   ↓

2. Loop until a solution is found or until a complete iteration produces no change to Current state:

(i) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.

↓

(ii) For each operator that applies to the current state do :

(a) Apply the operator and generate a new state.

(b) Evaluate the new state.

- If it is a goal state, then return it and quit.
- If not, compare it to SUCC. If it is better, then set SUCC to this state.
- If it is not better, leave SUCC alone.

(c) If the SUCC is better than current state, then set current state to SUCC.

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all **perturbations of the initial state and choose the best**. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-accent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

3.15.2 Limitations of Steepest-ascent Hill Climbing

UQ. State limitations of steepest-ascent hill climbing (MU - Q. 2(b), May 19, 5 Marks, Q. 6(b), May 18, 5 Marks, Q. 2(A), Dec. 17, 10 Marks, Q. 2(b), May 17, 10 Marks)

UQ. What are the problems/frustrations that occur in hill climbing technique? Illustrate with an example. (MU - Q. 4(b), Dec. 15, 6 Marks, Q. 1(d), May 17, 5 Marks)

Steepest-ascent hill climbing may fail to find a solution. Algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

1. **A local maximum** is a state that is better than all its neighbours but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called **foothills**.
2. **A plateau** is a flat area of the search space in which a whole set of neighboring 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.
3. **A ridge** is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

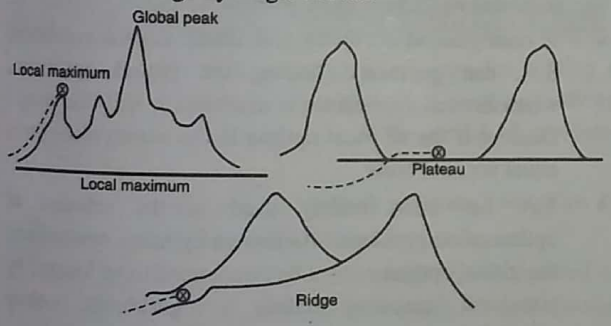


Fig. 3.15.1 : Problems associated with hill climbing : local maxima, plateau and ridge

3.15.3 Ways of dealing with Local Maxima Plateau and Ridge Problems

GQ. What are the ways of dealing with local maxima plateau and ridge problems which arise in hill climbing?

Problems associated with hill climbing are :

(i) Local maxima, (ii) plateaus, (iii) ridges.

There are some ways of dealing with these problems, although these methods are by no means guaranteed :

- Some problem spaces are great for hill climbing and others are terrible.
- **Random restart :** Keep restarting the search from random locations until a goal is found.
- **Problem reformulation :** Reformulate the search space to eliminate these problematic features.
- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.
- Make a big jump in some direction to try to get to a new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.
- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.
- Even with these first-aid measures, hill climbing is not always very effective. It is particularly unsuited to problems where the value of the heuristic function drops off suddenly as we move away from a solution. This is often the case whenever any sort of threshold effect is present.
- Hill climbing is a local method, by which we mean that it decides what to do next by looking only at the "immediate" consequences of its choice rather than by exhaustively exploring all the consequences. It shares with other local methods, the advantage of being less combinatorically explosive than comparable global methods. But it also shares with other local methods a lack of a guarantee that it will be effective. Although it is true that the hill-climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function.

Module 3

3.16 SIMULATED ANNEALING (SA)

GQ. Define the term simulated annealing. Explain simulated Annealing with suitable example.

MU - Q. 2(A), Dec. 17, 10 Marks, Q. 2(b), May 16, 10 Marks

- Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some **downhill moves** may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively intensive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau or a ridge.
- Simulated annealing as a computational process is patterned after the **physical process of annealing**, in which physical substances such as metals are melted (i.e., raised to high energy levels) and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-energy final state. Thus this process is one of valley descending in which the objective function is the energy level.
- Simulated annealing is an **effective and general form of optimisation**. Annealing refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal. Simulated annealing uses the objective function of an optimisation problem instead of the energy of a material.

Uses of SA

SA is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a met heuristic to approximate global optimisation in a large search space for an optimisation problem.

3.16.1 Types and Use of Simulated Annealing

- Simulated annealing algorithms are essentially random search methods in which the new solutions, generated according to a sequence of **probability distribution** (for example, the Boltzmann distribution) or a random procedure (e.g. a hit-and-run algorithm) may be accepted even if they do not lead to an improvement in the Annealing.
- Simulated Annealing is a process where the temperature is reduced slowly; starting from a random search at high temperature and then eventually it became purely greedy descent as it approaches zero temperature. S.A. maintains a current assignment of values of variables.
- Simulated Annealing is an effective and general form of optimisation. It is useful in finding global optima in the presence of large number of local optima. It is analogous to temperature in an annealing system. At higher value of T, uphill moves are more likely to occur.
- S.A. will accept an increase in the **cost function** with some probability based on annealing algorithm. Simulated Annealing is based on an analogy to a physical system which is first melted and then cooled or annealed into a low energy state.

3.16.2 Simulated Annealing in Machine Learning

- S.A. is a technique that is used to find the best solution for either a **global minimum or maximum** without having a check for every single possible solution, that exists. This is super helpful when addressing massive optimisation problems like the one previously stated.
- S.A. is a **stochastic global search optimisation algorithm**. The algorithm is inspired by annealing in metallurgy where metal is heated to a high temperature quickly, then cooled slowly and that increases its strength and makes it easier to work with S.A. executes the search in the same way.
- Annealing is a heat treatment process that changes the 'physical and sometimes' also the chemical properties of a material to increase ductility and reduce the hardness to make it more workable.
- It configurated correctly and under certain condition, S.A. can guarantee finding the **global optimum**, whereas such a guarantee is available to Hill Climbing / Descent if the all local optima in the search place have equal scores / costs.
- S.A. has been widely used in the solution of optimisation problems. As known by many researchers, the global optima cannot be guaranteed to be located by simulated annealing unless a logarithmic cooling schedule is used.

3.17 PARAMETER FOR S.A.

1. Choice of parameters depends on the **expected variation in the performance** measure over the search space.
2. A good rule of thumb is that the initial temperature should be set to accept roughly **98°** of the moves and that the final temperature should be low enough that the solution does not improve much, if at all. To improve simulated annealing, we have to do the following:
   (i) Improve the accuracy.
   (ii) Alter the parameters of the algorithm.
   (iii) You could run your own meta-optimisation on the parameters of your problem.

3. Simulated annealing is gradient based. It is an **extension of gradient descent**, and in the degenerate case (zero temperature) they are the same: It generates **random neighbouring states**, and if the fitness of that state is better than the current one then it jumps there. That is, it seeks a local minimum.
4. In annealing, atoms migrate in the **crystal lattice** and the number of dislocations decreases, leading to a change in ductility and hardness. With the knowledge of composition and phase diagram, heat treatment can be used to adjust from harder and more brittle to softer and more ductile.

Examples of SA

- Travelling salesman problem.
- Task allocation
- Graph colouring and partitioning
- Scheduling problems
- Non-linear function optimization

3.18 GENETIC ALGORITHM

UQ. Write a short note on genetic algorithm.
(MU - Q. 5(a), Dec. 15, Q. 6, May 17, 10 Marks)

- An algorithm is a progression of steps for solving a problem. A genetic algorithm is a problem-solving technique that uses genetics as its model of problem-solving.
- It is a search method to find estimated solutions to optimization and search issues.
- The genetic algorithm is a method that is applied to solve both constrained and unconstrained optimisation problems.
- It is based on natural selection.
- The genetic algorithm modifies a population of individual solutions.
- Genetic algorithm is meta heuristic in computer science and operations research.
- It is inspired by natural selection. And it belongs to the larger class of evolutionary algorithms.
- Genetic algorithms are used to generate high quality solutions to optimisation and search problems.
- It depends on biologically inspired operators such as crossover, mutation and selection. Some examples of Genetic Algorithm (GA) applications include optimising decision trees for better performance, hyper parameter optimisation etc.
- One can easily distinguish between a traditional and a genetic algorithm.

3.18.1 Comparison between Traditional and Genetic Algorithm

Traditional Algorithm	Genetic Algorithm
It selects the next point in the series by a deterministic computation.	It selects the next population by computation, which utilizes random number generators.
It creates an individual point at each iteration. The sequence of points approaches an optimal solution.	It creates a population of points atevery iteration. The best point in the population approaches an optimal solution.
Advancement in each iteration is problem specific.	Concurrence in each iteration is a problem independent

3.18.2 Basic Terminology

UQ. Define the terms chromosome, fitness function, crossover and mutation as used in Genetic algorithms. (MU - Q. 5(a), May 18, 10 Marks)

UQ. Explain how genetic algorithms work. Define the terms chromosome, fitness function, crossover and mutation as used in Genetic algorithms.
(MU - Q. 6(c), Dec. 18, 5 Marks)

We introduce ourselves to basic terminology required to understand GAs.

(i) **Population :** The population of GA is same as the population for human beings except that instead of human beings, we have **candidate solutions** that are similar to human beings.

(ii) **Chromosomes :** Chromosome is regarded as solution to the given problem.

(iii) **Gene :** A gene is one element position (or part) of a chromosome.

(iv) **Allele :** It is the value taken by a gene for a particular chromosome.

Module 3

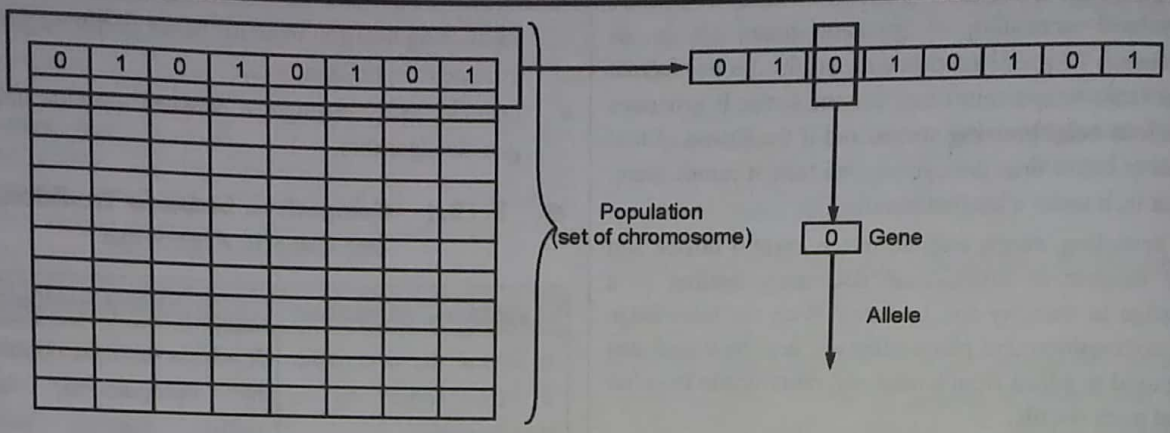


Fig. 3.18.1

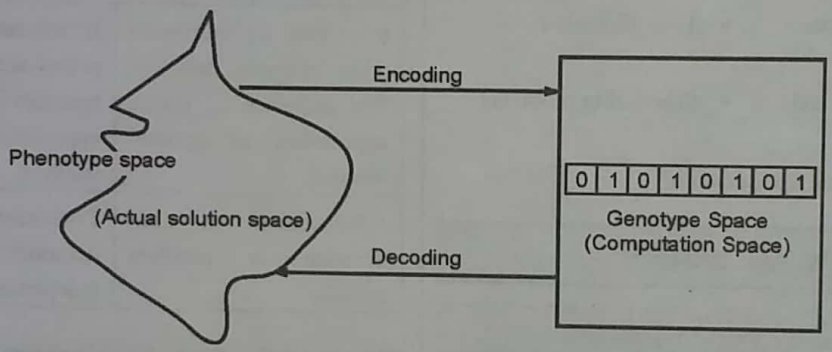


Fig. 3.18.2

(v) **Genotype :** Genotype is the population in the computation space. Using computing system, solutions represented in the computation space can be easily understood and easily manipulated.

(vi) **Phenotype :** Phenotype is the population in which solution are represented in a way they are represented in actual world situations.

(vii) **Decoding and Encoding :** Decoding is a process of transforming a solution from the genotype to the phenotype space.

Encoding is process of transforming from the phenotype to genotype space.

Decoding has to be fast as it is carried out repeatedly in a GA during the fitness value calculation.

Remark : For **simple problems** the phenotype and genotype spaces are **same**.

Fitness Function : A fitness function is a function which takes the solution as input and produces the suitability of the solution as the output.

In some cases the fitness function and objective functions are same. Depending on the problem, they may be different.

Fitness function is a function, we want to optimise.

3.18.3 Optimisation Problems

- In a genetic algorithm, a population of candidate solutions to an optimisation problem is evolved, towards better solutions.
- The evolution generally begins with a population of randomly generated individuals. It is an **iterative process,** and the population in each iteration called a generation.
- The fitness of every individual in the population is evaluated. The fitness of every individual is the valve of the **objective function** in the optimisation problem, to be solved.

The more fit individuals are stochastically selected from the current populations to form a new generation. And it is used in the next iteration of the algorithm. The algorithm terminates if (i) a maximum number of generations has been produced or (ii) a satisfactory fitness level is reached for the population.

Requirements of genetic algorithm are :

(i) a genetic representation of the solution domain,

(ii) a fitness function to evaluate the solution domain.

A standard representation of each candidate solution is an array of **bits.**

The main property that makes these genetic representation possible is that their parts are easily aligned due to their fixed size, which causes simple crossover operations.

- Once the genetic representation and the fitness function are defined, a genetic Algorithm initialises a population of solutions and then to improve it through repetitive application of the **mutation crossover, inversion** and **selection operators.**

3.18.4 Initialisation

- The population size depends on the nature of the problem. And it may have several thousands of solutions. Generally, initial population is generated randomly, and it may have all the possible solutions. (This is termed as **search space**).
- The solutions may be 'seeded' in areas where **optimal** solution may be found.

3.18.5 Selection

- During each successive generation, a portion of the existing population is selected to obtain a new generation.
- Individual solutions are selected through a fitness-based process. Certain selection methods rate the fitness of each solution and select the best solution.
- The fitness function is defined over genetic representation. It measures the quality of the represented solution. The fitness function is always problem dependent.
- In some problems, it is not possible to define the fitness expression, in such cases, simulation may be used to determine the fitness function valve of a phenotype (for example, computational fluid dynamics is used to determine the air resistance of a vehicle whose shape is encoded as the **interactive genetic algorithms).**

3.18.6 Genetic Operators

UQ. Explain how genetic algorithms work.
(MU - Q. 6(c), Dec. 18, 5 Marks)

Genetic operators alter the genetic composition of the offspring.

These include crossover, mutation, selection, etc.

Basic structure : The basic structure of GA is

(i) We begin with an initial population, it may be generated at random or seeded by other heuristics.

(ii) Select parents from this population for mafing,

(iii) Apply crossover and mutation operators on the parents to generate new off-springs.

(iv) Finally these off-springs replace the existing individuals in the population and

(v) Process gets repeated.

- In this way genetic algorithms actually copy the human evolution to some extent.

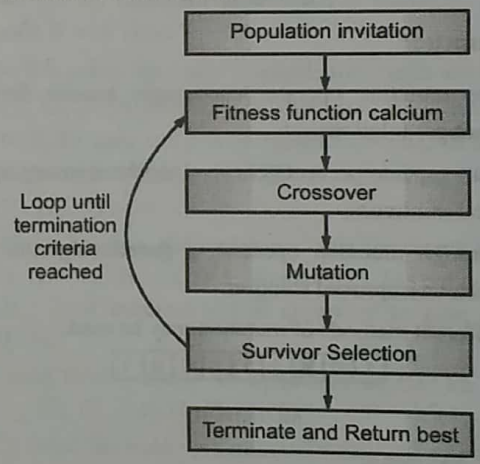


Fig. 3.18.3 : Basic Structure of Genetic Operator

- Genetic variation is a necessity for the process of evolution.
- Genetic operators used in genetic algorithms are analogous to those in the natural world: selection, crossover (also called recombination) and mutation.

(1) Selection

(i) Selection operators give preference to better solutions (chromosomes).

(ii) Allowing them to pass on their 'genes' to the next generation of the algorithm.

Module 3

(iii) The best solutions are determined using some form of **objective function** (also known as a 'fitness function' in genetic algorithms), before being passed to the crossover operator.

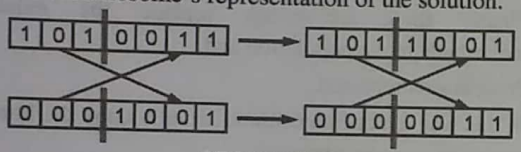
(2) Crossover

(i) Crossover is the process of taking more than one **parent solutions** (chromosomes) and

(ii) Producing a child solution from them. By recombining portions of good solutions.

(iii) The genetic algorithm may create a better solution.

(iv) The crossover method is often chosen to closely match the chromosome's representation of the solution.



CROSSOVER

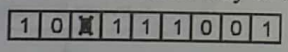
(3) Mutation

(i) The mutation operator encourages genetic diversity amongst solutions,

(ii) It attempts to prevent genetic algorithm converging to a local minimum,

(iii) Through mutation operator, a genetic algorithm can reach an improved solution.

(iv) Different methods of mutation may be used.



MUTATION

(4) Termination

- The above process is repeated until a termination condition is reached.
- In general, terminating conditions are :
  1. A solution is found that satisfies minimum criteria.
  2. Fixed number of generation reached.
  3. computation time/allocated money reached.
  4. The solution's fitness is totally reached or any further successive iteration no longer produce better results.
  5. Manual inspection and combination of the above conditions.

3.18.7 Advantages of Genetic Algorithm

1. The genetic algorithm concept is easy to understand.
2. The genetic algorithm supports multi-objective optimization.
3. A genetic algorithm is suitable for noisy environments.
4. The genetic algorithm is robust with respect to local minima/maxima.
5. The genetic algorithm utilizes probabilistic transition rules.
6. The genetic algorithm utilizes payoff (objective function) information, not derivatives.
7. The genetic algorithm works well on mixed discrete functions.
8. The genetic algorithm concept is modular, separate from application.
9. In the genetic algorithm concept, answer gets better with time.
10. The genetic algorithm concept is inherently parallel; easily distributed.
11. The genetic algorithms work on the Chromosome, which is an encoded version of potential solutions' parameters, rather the parameters themselves.
12. The genetic algorithms use fitness score, which is obtained from objective functions, without other derivative or auxiliary information

3.18.8 Limitations of Genetic Algorithm

1. The Genetic Algorithms might be costly in computational terms since the evaluation of each individual requires the training of a model.
2. These algorithms can take a long time to converge since they have a stochastic nature.
3. The language used to determine candidate solutions must be robust. It must be able to endure random changes such that fatal errors don't mistake.
4. A wrong decision of the fitness function may lead to significant consequences
5. Small population size will not give enough solution to the genetic algorithm to produce precise results.
6. A frequency of genetic change or poor selection scheme will result in disrupting the beneficial schema.
7. Though Genetic algorithms can find exact solutions to analytical sorts of problems, traditional analytic techniques can find the same solutions in a short time with few computational data.

3.18.9 Applications of Genetic Algorithm

1. Genetic Algorithm in Robotics

- As we know Robotics is one of the most discussed fields in the computer industry today.
- It is used in various industries in order to increase profitability efficiency and accuracy.
- As the environment in which robots work with the time change, it becomes very tough for developers to figure out each possible behaviour of the robot in order to cope with the changes.
- This is the place where the Genetic Algorithm places a vital role.
- Hence a suitable method is required, which will lead the robot to its objective and will make it adaptive to new situations as it encounters them.
- Genetic Algorithms are adaptive search techniques that are used to learn high-performance knowledge structures.

2. Genetic Algorithm in Financial Planning

- Genetic algorithms are extremely efficient for financial modelling applications.
- As they are driven by adjustments that can be used to improve the efficiency of predictions and return over the benchmark set.
- In addition, these methods are robust, permitting a greater range of extensions and constraints, which may not be accommodated in traditional techniques.

3.19 BEST FIRST SEARCH

GQ. Explain best first search with algorithm.
OR When best first search algorithm, will be applicable ? With a suitable algorithm and example explain the best first search.

❑ **Definition :** This search procedure is an evaluation function variant of best the first search. The heuristic function used here called an evaluation function is an indicator how far the node is from the goal node. Goal nodes have an evaluation function value of zero.

Best-first search is explained using a search graph given in figure :

1. First, the start node S is expanded. It has three children A, B and C with values 3, 6 and 5 respectively. These values approximately indicate how far they are from the goal node.
2. The child with minimum value namely A is chosen. The children of A are generated. They are D and E with values 9 and 8.
3. The search process has now four nodes to search for. i.e., node D with value 9, node E with value 8, node B with value 6 and node C with value 5. Of them, node C has got the minimal value which is expanded to give node H with value 7.
4. At this point, the nodes available for search are (D : 9), (E : 8), (B : 6) and (H : 7) where (α : β) indicates that (α) is the node and β is its evaluation value. Of these, B is minimal and hence B is expanded to give (F : 12), (G : 14).
5. At this juncture, the nodes available for search are (D : 9), (E : 8), (H : 7), (F : 12) and (G : 14) out of which (H : 7) is minimal and is expanded to give (I : 5), (J : 6).
6. Nodes now available for expansion are (D : 9), (E : 8), (F : 12), (G : 14), (I : 5), (J : 6).

   Of these, the node with minimal value is (I : 5) which is expanded to give the goal node.

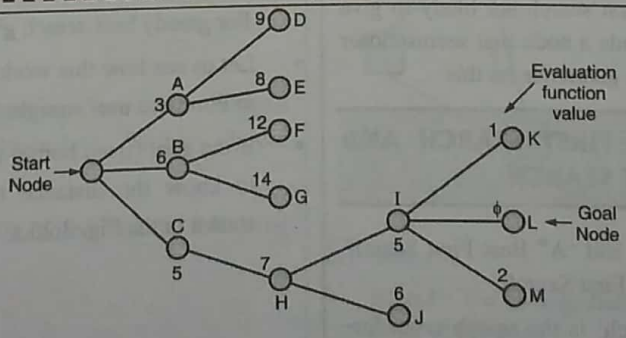


Fig. 3.19.1 : A sample tree for best-first search

Module 3

3.19.1 Steps of the Search Process in BFS

The entire steps of the search process are given in Table 3.19.1

Table 3.19.1 : Search process of best-first search

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

As you can see, best-first search "**jumps all around**" in the search graph to identify the node with minimal evaluation function value. There is only a minor variation between hill-climbing and best-first search. In the former, we sorted the children of the first node being generated. Here, we have to sort the entire list to identify the next node to be expanded.

3.19.2 Algorithm for best-first Search

- **Step 1 :** Put the initial node on a list START.
- **Step 2 :** If (START is empty) or (START = GOAL), then terminate search.
- **Step 3 :** Remove the first node from START. Call this as node *a*.
- **Step 4 :** If (*a* = GOAL), then terminate search with success.
- **Step 5 :** Else if node *a* has successors, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from the goal.
- **Step 6 :** Name this list as START 1.
- **Step 7 :** Replace START with START 1.
- **Step 8 :** Goto Step 2.

The paths found by best-first search are likely to give solutions faster because it expands a node that seems closer to the goal. However, there is no guarantee on this.

3.20 GREEDY BEST FIRST SEARCH AND A* BEST FIRST SEARCH

- 'Greedy Best First Search' and 'A* Best First Search' are the two variants of Best First Search.
- The 'Greedy best first search' is the search procedure and is evaluation function variant of breadth first search. The heuristic function used here is called an evaluation function and is an indicator how far the node is from the goal node. Goal nodes have an evaluation function of valve zero.
- Speaking about the greedy and best first search, it is the 'best' first search that is named as greedy best first search only.
- But there is a family of best first search algorithms, which have different evaluation functions. The best first search makes use of the function g (n) along with h (n).
- The evaluation function f (n) represents the total cost. Here f (n) = g (n) + h (n), where g (n) is the cost so far to reach n, while h (n) is the estimated cost from n to the goal state.

  For greedy best-search g (n) = 0, and f (n) = h (n)
- Let us see how this works for routine-finding problems in Pune, we use 'straight-line distance' heuristic.
- If the goal (from home) to Roopali restaurant, we need to know the distance to Roopali Hotel, which are shown in the Fig. 3.20.1.

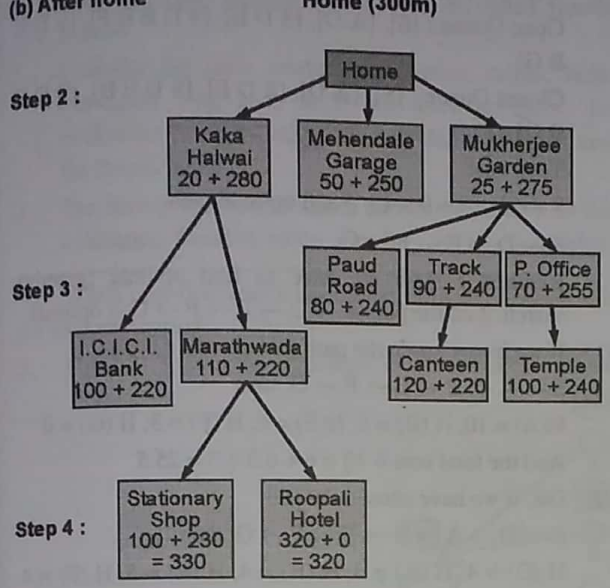


Fig. 3.20.1

- The straight line distance heuristic, we denote by h_{SLD}. Now, h_{SLD} cannot be calculated from the problem description itself. It takes a little experience to know that h_{SLD} is approximately equal to actual road distance, and hence it is a useful heuristic.

Kaka Halwai – 20 meters

Mehendale Garage – 40 meters

Mukherjee Garden – 25 meters

I.C.I.C.I Bank – 90 meters

B.M.C.C. College – 120 meters

Marathwada College – 140 meters

Stationary shop – 180 meters

Roopali Hotel – 300 meters

School – 100 meters

Walking track – 110 meters

Post Office – 130 meters

Stages in a greedy best-first tree search for Roopali Hotel with the straight line distance heuristic h_{SLD}. Nodes are labelled with distances.

For the algorithm, a node should contain the following information :

1. Description of the state.
2. Its parent link
3. List of nodes that are generated from it.

A graph containing these information is called 'OR' graph.

3.20.1 Greedy Best First Search Algorithm

Now we discuss the algorithm :

1. Initial state-node.
2. Proceed (till goal is found).
   (i) Select the best node. Add to the list.
   (ii) Mark the successors.
   (iii) For every successor,
      (a) Since priority queue is used, select the best node.
      (b) If there is already successor, change the parent if this new path is better than the earlier one. And update the costs to this node.

   If the successor is not generated then evaluate the cost on the basis of heuristic.
3. Repeat the process.

Properties of Greedy Best-First Search

(i) Greedy Best-First Search is not optimal.

(ii) It is incomplete, even if the given state consists of finite number of nodes.

3.20.2 Solved Examples

Ex. 3.20.1 : Consider the following graph shown in the Fig. Ex. 3.20.1 is the starting state, G is the goal state. Run the greedy search algorithm and write the order of the node in which it is explored. The straight line distance heuristic estimate for the nodes are also given below :

h(s) = 10.5, h (A) = 10, h(B) = 6, h(c) = 4,

h(D) = 8, h(E) = 6.5, h(F) = 3 and h(G) = 0

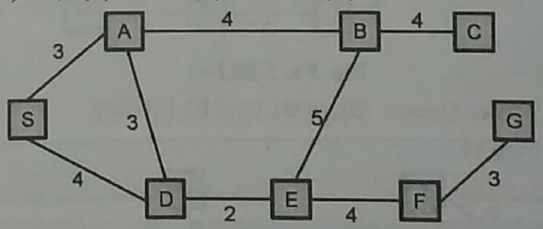


Fig. Ex. 3.20.1

Soln. :

Step I : We have to find greedy search. We choose the order of the node

Consider the tree

Module 3

(i) **Cost :** F (A) = H (A) = 10

F(D) = H (D) = 8

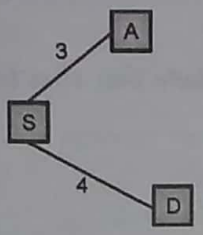


Fig. Ex. 3.20.1(a)

(ii) W closed [S] and [S D] on closed queue and [A D] open Queen.

▶ **Step II :** We consider the tree A S D E

(i) Cost : F (E) = H (E) = 6.5 and closed Queen [S], [S D], [S D E] and open queue : [S], [A D], [S, D, E]

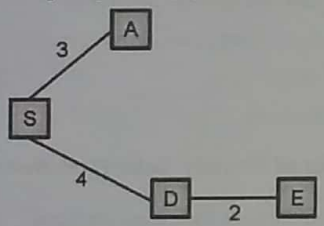


Fig. Ex. 3.20.1(b)

▶ **Step III :**

(i) **Cost :** F (B) = H (B) = 6

F(B) = H (F) = 3

Open Queue : [S], [A D], [S D E], [S D E B F]

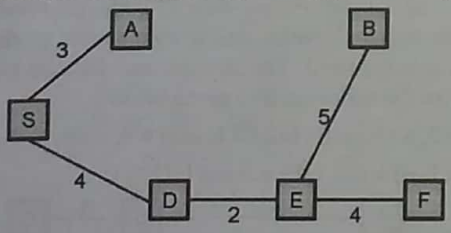


Fig. Ex. 3.20.1(c)

Closed Queue : [S], [S D], [S D E], [SDEB]

▶ **Step IV :**

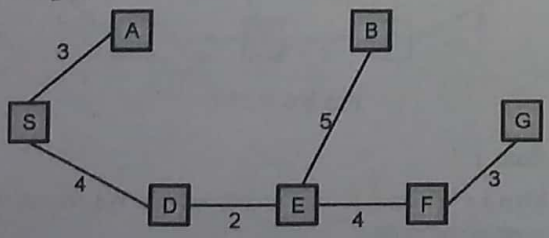


Fig. Ex. 3.20.1(d)

(i) **Cost :** F(G) = H (G) = 0

Open Queue : [S], [A D], [S D E], [S D E B F], [S D E B G]

Closed Queue : [S], [A D], [S D E], [S D E B], [S D E B G]

Total cost

8 + 6.5 + 3 + 0 = 17. 5 and optimal path

S → D → E → F → G

Remark : Here we have to find optimal (greedy) search. And the path S → D → E → F → G is optimal

(1) If we have chosen the path

S → A → B → E → F → G, then

H(A) = 10, H (B) = 6, H(E) = 6, H (F) = 3; H (G) = 0

And the total cost = 10 + 6 + 6.5 + 3 = 25.5

(2) OR, if we have chosen the path

S → D → A → B → E → F → G; then

H (D) = 4, H (A) = 3, H (B) = 4, H (E) = 5, H (F) = 4 and H (G) = 0

Then the total cost = 4 + 3 + 4 + 5 + 4 = 21

(3) For any other path, the total cost would have been greater than 17. 5

∴ S → D → E → F → G is optimal path and optimal cost is 17.5

3.21 A* AND AO* SEARCH

UQ. Explain A* Algorithm with example.

(MU - Q. 2(b), May 18, 10 Marks. (MU - Q. 2(b), May 19, 5 Marks. Q. 6(c), May 18, 5 Marks. Q. 6(b), May 17, 5 Marks)

1. **A* Algorithm :** In best-first search, we brought in a heuristic value called **evaluation function value**. It is a value that estimates how far a particular node is from the goal. Apart from the evaluation function value, one can also bring in cost functions. Cost functions indicate how much resource like time, energy, money etc. have been spent in reaching a particular node from the start. While evaluation function values deal with the future, cost function values deal with the past. Since cost function values are really expended, they are more concrete than evaluation function values.
2. If it is possible for one to obtain the evaluation function values and the cost function values, then A* algorithm can be used. The basic principle is that sum the cost and evaluation function value for a state to get its "**goodness**" worth and use this as a **yardstick** instead of the evaluation function value in best-first search. The sum of the evaluation function value and the cost

along the path leading to that state is called fitness number.

3. Consider the same evaluation function values. Now associated with each node are three numbers, the evaluation function value, the cost function value and the fitness number.
4. The fitness number, as stated earlier, is the total of the evaluation function value and the cost-function value.

For example, consider node K, the fitness number is 20, which is obtained as follows :

(Evaluation function of K) + (Cost function involved from start node S to node K)

= 1 + (Cost function from S to C + Cost function from C to H + Cost function from H to I + Cost function from I to K)

= 1 + 6 + 5 + 7 + 1 = 20.

While best-first search uses the evaluation function value only for expanding the best node, A* uses the fitness number for its computation.

5. Fig. 3.21.1 gives the algorithm for A* algorithm method.

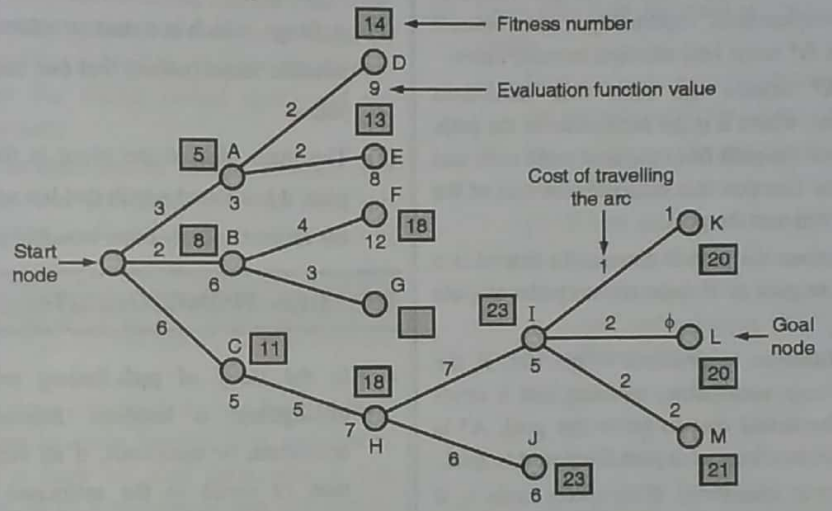


Fig. 3.21.1 : Sample tree with fitness number used for A* search

- **Step 1 :** Put the initial node on a list START.
- **Step 2 :** If (START is empty) or (START = GOAL), then terminate search.
- **Step 3 :** Remove the first node from START. Call this as node a.
- **Step 4 :** If (a = GOAL), then terminate search with success.
- **Step 5 :** Else if node a has successors, generate all of them. Estimate the fitness number of the successors by totaling the evaluation function value and the cost-function value. Sort the list by fitness number.
- **Step 6 :** Name this new list as START 1.
- **Step 7 :** Replace START with START 1.
- **Step 8 :** Go to Step 2.

3.22 ADMISSIBILITY OF A*

UQ. Prove the admissibility of A*. (MU - Q. 5(B), Dec. 16, 6 Marks)

- Admissible functions are functions that should satisfy the essential boundary conditions of the problem.
- An algorithm A is admissible if it is guaranteed to return an optimal solution when one exists.
- A* algorithm is admissible if, it uses an admissible heuristic', and h(goal) = 0 (h(n) is smaller than h*(n)), then A* is guaranteed to find an optimal solution, i.e., f(n) is non-decreasing along any path.

Module 3

[**Theorem :** If h(n) is consistent, f along any path is non-decreasing]

- If is heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least cost path from start to goal.
- Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand.
- A* is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. Thus A* is the best solution in many cases.
- Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal.
- A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.
- The heuristic function is problem – specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.
- A heuristic h(n) is admissible if for every node n, if $h(n) < h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- **Theorem :** A* is optimal if two conditions are met.
  (1) The heuristic is admissible, as it will never overestimate the cost.
  (2) The heuristic is monotonic, that is, $h(n_i) < h(n_{i+1})$, then real-cost (n_i) < real – cost (n_{i+1}).

Proof : Let G be an optimal goal and let n be an unexpanded node in the fringe such that n is on a shortest path.

Let some suboptimal goal G_1 has been generated and is in the fringe.

Now, we have

$$f(G_1) = g(G_1) + h(G_1)$$

$$\therefore \quad f(G_1) = g(G_1)$$

[$\because h(G_1) = 0$, $\because G_1$ is in the fringe and is an optimal goal]

Again $f(G) = g(G)$

$\because h(G) = 0$

Now, $f(G_1) > f(G)$ [$\because$ heuristic is monotonic] ...(i)

$\therefore \quad h(n) \leq h^*(n)$ [$\because$ h is admissible]

$\therefore g(n) + h(n) \leq g(n) + h^*(n)$

$\therefore f(n) \leq f(G) < f(G_1)$ ...From equation (ii)

$\therefore f(G_1) > f(n)$,

$\therefore$ A* will never select G_1 for expansion.

Remarks

(1) A fringe, which is a data structure used to store all the possible states (nodes) that one can go from the current states.

(2) The main idea of the proof is that when A* finds a path, it has found a path that has an estimate lower than the estimate of any other possible path.

3.23 MONOTONICITY

- In the study of path-finding problems in artificial intelligence, a heuristic function is said to be consistent, or monotone, if its estimate is always less than or equal to the estimated distance from any neighboring vertex to the goal, plus the cost of reaching that neighbor.
- Thus, for every node n and each successor p of n, the estimated cost of reaching the goal from n is not greater than the step cost of getting to P plus the estimated cost of reaching the goal from P. That is $h(n) \leq C(n, p) + h(p)$ and $h(G) = 0$.

Where :

h is the consistent heuristic function :

n is any node in the graph,

p is any descendant of n

G is any goal node

C(n, p) is the cost of reaching node p to n.

Hence, every node i will give an estimate that accounting for the cost to reach the next node, is always lesser than the estimate at node (i + 1).

Thus monotonicity is established.

3.23.1 Monotonicity in AI

Monotonicity in AI can refer to monotonic classification or monotonic reasoning. Monotonic classification is a mathematical property of an AI model closely related to the concept of a monotonic function. Monotonic reasoning is a form of reasoning that can underlie AI system's logic.

3.23.2 Concept

- A* algorithm is a searching algorithm that searches for the shortest path between the initial and the final state. It is used in various applications, such as maps. In maps the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).
- A* search is a combination of lowest cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand.
- It uses cost(p), the cost of the path found, as well as the heuristic function h(p), the estimated path cost from the end p to the goal.
- A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start node g(n). It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently.
- A* is complete and optimal on graphs that are locally finite where the heuristics are admissible and monotonic.
- Because A* is monotonic, the path cost increases as the node gets further from the root.
- Algorithm A* is a best-first search algorithm that relies on an open list and a closed list to find a path that is both optimal and complete towards the goal. It works by combining the benefits of the uniform-cost search and greedy-search algorithms. A* makes use of both elements by including two separate path finding functions in its algorithm that take into account the cost from the root node to the current node and estimates the path cost from the current node to the goal node.
- The first function is g(n), which calculates the path cost between the start node and the current node.
- The second function is h(n), which is a heuristic to calculate the estimated path cost from the current node to the goal node.

$$f(n) = g(n) + h(n).$$

- It represents the path cost of the most efficient estimated path towards the goal. A* continues to revaluate both g(n) and h(n) throughout the search for all of the nodes that it encounters in order to arrive at the minimal cost path to the goal. This algorithm is extremely popular for path finding in strategy computer games.

The process for A* is basically this :

(1) Create an open list and a closed list that are both empty. Put the start node in the open list.

(2) Loop this until the goal is found or the open list is empty.

  (a) Find the node with the lowest F cost in the open list and place it in the closed list.

  (b) Expand this node and for the adjacent nodes to this node.

    (i) If they are on the closed list, ignore.

    (ii) If not on the open list, add to open list, store the current node as the parent for this adjacent node, and calculate the F, G, H costs of the adjacent node.

    (iii) If on the open list, compare the G costs of this path to the node and the old path to the node. If G cost of using the current node to get the node is the lower cost, change the parent node of the adjacent node to the current node.

(3) If open list is empty, fail.

- The terms locally finite, admissible, and monotonic all aid in the understanding of when A* can be expected to be complete, meaning that it finds a solution, and optimal, meaning that it finds the solution with the lowest path cost.
- A locally finite graph is one where none of the nodes on the graph have an infinite branching factor, thus none of the node paths branch forever. A branching factor of a node refers to the amount of new nodes that can be expanded from that node.

UEx. 3.23.1 : (MU - Q. 2(a), Dec. 2015, 10 Marks)

Consider the graph given in the Fig. Ex. 3.22.1. 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 estimate 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

Module 3

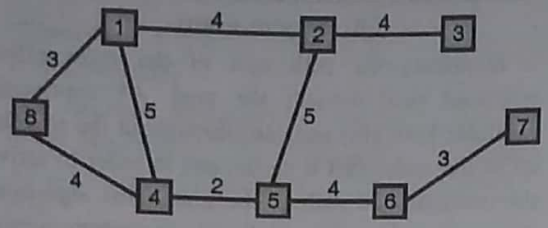


Fig. Ex. 3.23.1

☑ **Soln. :**

▶ **Step I :** We put the initial node S on a list START.

START is not empty and START ≠ GOAL

START : We generate the nodes ('.' It has successors)

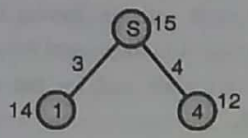


Fig. Ex. 3.23.1(a)

Now, open : 4(12 + 4), 1(14 + 3)

Closed : 5(15)

▶ **Step 2 :** We name the new list as START 1 and go to step (2)

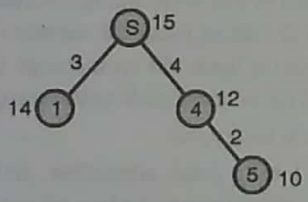


Fig. Ex. 3.23.1(b)

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

Closed : 5(15), 4 (12 + 4)

Remark :

We maintain two lists, open and closed. In the closed list, we place those nodes which have already expanded and in the open list, it places nodes which have yet not been expanded.

▶ **Step 3 :** We continue the process. We apply successors to node (5);

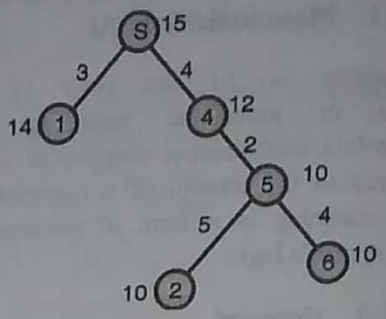


Fig. Ex. 3.23.1(c)

Again : open : 1 (14 + 3), 6(10 + 10), 2(10 + 11)

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

▶ **Step 4 :** We apply successors to node (1)

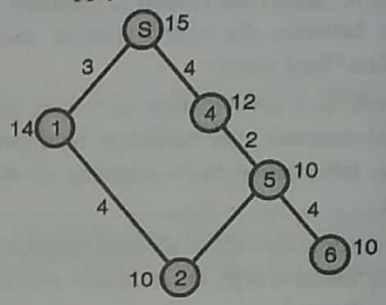


Fig. Ex. 3.23.1(d)

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

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

▶ **Step 5 :** We apply successor to node (2)

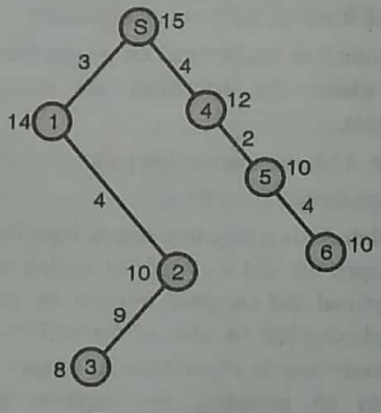


Fig. Ex. 3.23.1(e)

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

Closed :

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

▶ **Step 6 :** On simplification, step (5) becomes

Open : 6(10 + 10)

Closed :

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

Step 7 : We apply goal –node to sub node (6)

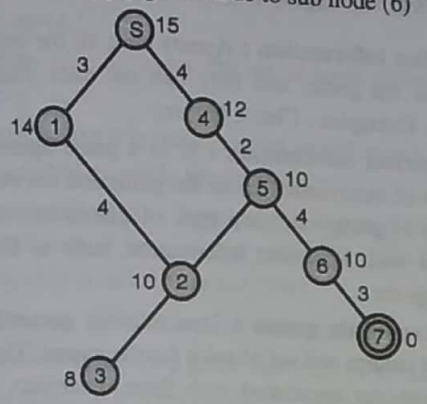


Fig. Ex. 3.23.1(f)

Open : 7 (0 + 13)

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

UQ. Compare following informed search algorithms
(a) Greedy first search
(b) A*
(c) Recursive Best-First (RBFS)
(MU- Q.4(a), May 16, 10 Marks)

(a) Greedy first search

(i) The Greedy best-first search algorithm tries to explore the node that is closest to the goal.

(ii) The algorithm evaluates nodes by using the heuristic function h(n)

(iii) That is , the evaluation function is equal to heuristic function,

f(n) = h(n). This equivalency is what makes the search algorithm 'greedy'.

(iv) The evaluation function of a target node is a weighted sum of the heuristic function and the distance from the current node to that target.

(v) It is the combination of depth-first search and breadth-first search algorithms.

(vi) It uses the heuristic function and search

(vii) The greedy best first algorithm is implemented by the priority queue.

(viii) The worst case **time complexity** of Greedy best first search is $O(b^m)$

(ix) **Space complexity :** The worst case space complexity of Greedy best first-search is $O(b^m)$, where m is the maximum depth of the search space.

(x) **Complete :** Greedy best-first search is also incomplete, even if the given space is finite.

(xi) **Optimal :** Greedy best first search is not optimal

(b) A*

(i) A* is a form of best-first search.

(ii) It uses heuristic function h(n).

(iii) Cost is to reach the node n from the start state g(n).

(iv) It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently.

(v) A* search algorithm finds the shortest path through the search space using the heuristic function.

(vi) A* algorithm search finds the shortest path through the search space (using heuristic function).

(vii) This algorithm can solve very complex problems.

(viii) **Time Complexity :** The time complexity of A* search algorithm depends on heuristic function. Time complexity is $O(b^{\square} d)$, where b is the branching factor.

(ix) **Space complexity :** The space complexity of A* search algorithm is $O(b^{\square} d)$.

(x) **Complete :** A* algorithm is complete as long as :
(i) Branching factor is finite; and
(ii) Cost at every action is fixed.

(xi) **Optimal :** A* search algorithm is optimal if it follows two conditions :
(i) h(n) should be an admissible heuristic for A* tree search.
(ii) Consistency for A* graph-search.

(c) Recursive Best-First (RBFS)

(i) It is simple recursive algorithm and resembles the operation of standard best first search.

(ii) It uses only linear space.

(iii) It is similar to recursive DFS.

(iv) It is more efficient than IDA*

(v) It suffers from excessive node regeneration.

(vi) It keeps track of the f value of the best alternative path available from any ancestor of the current node.

(vii) It uses the problem specific information about the environment to determine the preference of one node over the other.

(viii) **Time complexity :** Its time complexity is difficult to characterise because it depends on the accuracy of h(n) and as the nodes are expanded, the best path changes.

Module 3

(ix) **Space complexity :**
Space complexity is O(bd)

(x) **Complete :** Yes similar to A*

(xi) It is an optimal algorithm if h(n) is admissible; i.e. it is asymptotically optimal.

3.24 ADVERSARIAL SEARCH

- Adversarial search is a search when there is an **'enemy'** or **'opponent'** changing the state of the problem every step in a direction which we do not want to have.
- Each agent needs to consider the action of other agent and effect of that action on their performance. So, searchers in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called **adversarial searches**, often known as Games.
- Adversarial search is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want.
- **Examples :** Chess, business, trading, war. You change state, but then you do not control the next state. Opponent will change the next state in a way
  1. Unpredictable 2. hostile to you

  You only have to change every alternate state.

In adversarial search we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

(i) We study situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.

(ii) The environment with more than one agent is termed as multi-agent environment in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action other agent and effect of that action on their performance.

(iii) searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as 'Games'.

(iv) Games are modelled as a search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

3.24.1 Types of Games in AI

(i) **Perfect information :** Agents have all the information about the game, and they can see each other moves also. Examples : Chess, Go etc.

(ii) **Imperfect information :** If in a game agents do not have all information about the game and not aware with what is going on, such type of games are called the game with imperfect information, such as Battleship, bridge etc.

(iii) **Deterministic games :** Deterministic games follow a strict pattern and set of rules for the games. There is no randomness associated with them. Examples : Chess, Tic-Tac-Toe etc.

(iv) **Non-deterministic games :** These are those games which have various unpredictable events and have a factor of chance or luck. These are random, and each action response is not fixed. Such games are called as **stochastic games**. Example : Poker etc.

(v) **Zero-sum Game :** Zero-sum games are adversarial search which involves pure competition.

In zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.

One player of the game tries to maximise one single value, while other player tries to minimise it.

Each move by one player in the game is called as 'ply'.

Chess and Tic-Tac-Toe are examples of a zero-sum game.

Zero-sum game : Embedded thinking :

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out

(i) What to do

(ii) How to decide the move

(iii) Needs to think about his opponent as well

(iv) The opponent also thinks what to do. Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem

A game can be defined as a type of search in AI which can be formalised of the following elements :

(i) **Initial state :** It specifies how the game is set up at the start.

(ii) **Player (s) :** It specifies which player has moved in the state space.

(iii) **Actions (s) :** It returns the set of legal moves in state space.

(iv) **Result (s, a) :** It is the transition model, which specifies the result of moves in the state-space

(v) **Terminal-Test (s) :** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.

(vi) **Utility (s, p) :** A utility function gives the final numerical value for a game that ends in terminal states S for player p. It is also called payoff function.

Non-zero sum game : In non-zero sum game, each agent's gain or loss is not balanced by loss or gain.

If player wins the game, then it does not mean that the other player has lost the game.

☞ **Positive sum game**

Here all players have the same goal and they contribute together to play the game.

☞ **Negative sum game**

Here nobody wins, everybody loses. Every player has a different goal; e.g. war

Game-Tree

A game tree is a tree where nodes of the tree are the game states and edges of the tree are the moves by players. Game tree involves initial state, actions function and result function.

Example : Tic-Tac-Toe game tree

3.24.2 Characteristics of Adversarial Search (A.S.)

- Adversarial search in Artificial Intelligence is a game playing technique, where the agents are surrounded by a competative environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game.
- Searchers in which two or more players with conflicting goals are trying to explore the same 'search space' for the solution is called Adversarial Search.
- In A.S. there is enemy or opponent : Example : Chess, business, trading, war.
- Here an agent can change the state but then it cannot control the next state. Opponent will change the next state in an unprecedented way. Each agent needs to consider the action of other agent and effect of that action on its performance.

Module 3

3.24.3 Comparison of Search and Games

Sr. No.	Search	Games
1.	Adversarial search is a search where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.	Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called Adversarial searches, often known as Games.
2.	To find optimal solution Heuristics techniques are used. In adversarial search, the result depends on the players which will decide the result of the game.	Games are modelled as a search problem and heuristic evalution function, and these are the two main factors which help to model games in AI.
3.	Pruning is a technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.	**Perfect information :** A game with the perfect information is that in which agents can look into the complete board. Examples are chess, Go etc.
4.	Heuristic evaluation function allows to approximate the cost value at each level of the search tree, before reaching the goal node. Examples are : Chess business trading war you can change the state but then you cannot	**Imperfect information :** If in a game agents do not have all information about the game and not aware with what is going on, such type of games are called imperfect information. **Example :** Bridge, Battleship blind etc.

Sr. No.	Search	Games
	• NOTES •	**Deterministic games :** They follow a strict pattern and set of rules for the games. For example : Chess tic-tac-toe. **Non-deterministic games :** Such games have various unpredictable events and has a factor of change or luck. These are random, and each action response is not fixed. **Example :** Monopoly, Poker etc.

3.25 TECHNIQUES REQUIRED TO GET THE BEST OPTIMAL SOLUTION

There is always a need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfil our requirements :

- **Pruning** : A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- **Heuristic Evaluation Function** : It allows to approximate the cost value at each level of the search tree, before reaching the goal node.

3.26 GAME PLAYING

3.26.1 Zero Sum Game

UQ. Write short note on : Game Playing.
(MU - Q. 6(a), May 17, 5 Marks, Q. 6(e), May 17, 5 Marks)

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.
- The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:
  - What to do.
  - How to decide the move
  - Needs to think about his opponent as well
  - The opponent also thinks what to do
- Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

3.26.2 Elements of Game Playing Search

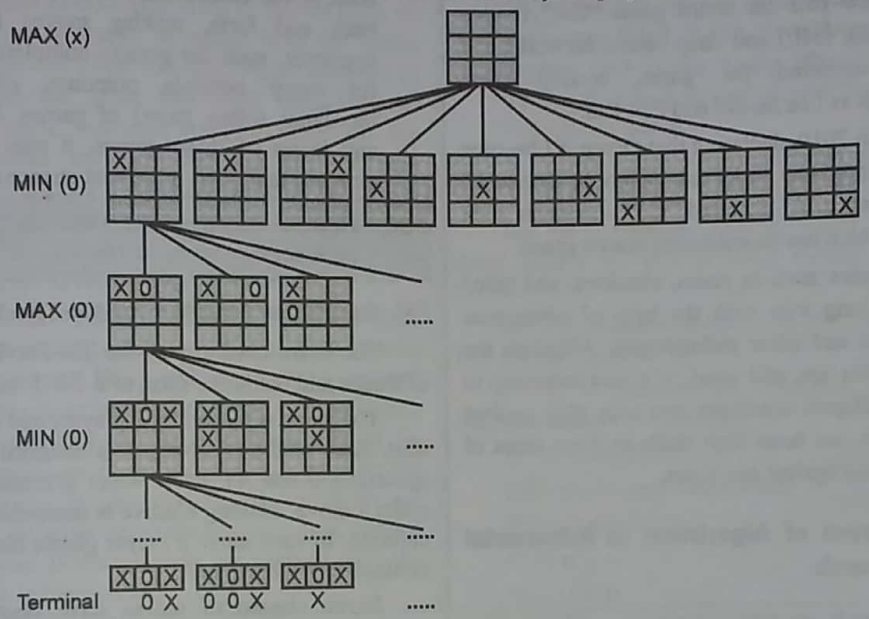
UQ. Draw a game tree for a tic-tac-toe problem.
(MU - Q. 4(c), Dec. 2015, 4 Marks)

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:

- **S0 :** It is the initial state from where a game begins.
- **PLAYER (s) :** It defines which player is having the current turn to make a move in the state.
- **ACTIONS (s) :** It defines the set of legal moves to be used in a state.
- **RESULT (s, a) :** It is a transition model which defines the result of a move.
- **TERMINAL-TEST (s) :** It defines that the game has ended and returns true.
- **UTILITY (s,p) :** It defines the final value with which the game has ended. This function is also known as Objective function or Payoff function. The price which the winner will get i.e.
- **(-1) :** If the PLAYER loses.
- **(+1) :** If the PLAYER wins.
- **(0) :** If there is a draw between the PLAYERS.

For example, in chess, tic-tac-toe, we have two or three possible outcomes. Either to win, to lose, or to draw the match with values **+1, –1 or 0.**

Let's understand the working of the elements with the help of a game tree designed in Fig. 3.26.1 for tic-tac-toe. Here, the node represents the game state and edges represent the moves taken by the players.



(1B54)**Fig. 3.26.1 : A game-tree for tic-tac-toe**

- **INITIAL STATE (S_0) :** The top node in the game-tree represents the initial state in the tree and shows all the possible choice to pick out one.
- **PLAYER (s) :** There are two players, **MAX and MIN. MAX** begins the game by picking one best move and place **X** in the empty square box.
- **ACTIONS (s) :** Both the players can make moves in the empty boxes chance by chance.
- **RESULT (s, a) :** The moves made by **MIN** and **MAX** will decide the outcome of the game.
- **TERMINAL-TEST(s) :** When all the empty boxes will be filled, it will be the terminating state of the game.
- **UTILITY :** At the end, we will get to know who wins: **MAX** or **MIN,** and accordingly, the price will be given to them.

3.26.3 Some More Examples of Game Playing/Adversarial Search

(1) **Chess :** In 1997, IBM's supercomputer dubbed "Deep Blue" did just that. And it didn't beat just any person, but the world chess champion Garry Kasparov. In an essay, Kasparov wrote about his first game with Deep Blue in 1996. He said that while he played numerous times against a computer, his match against Deep Blue was different. He sensed a "new kind of intelligence across the table." During a rematch in 1997, the chess master lost to the supercomputer. In this instance, we can deduce that Deep Blue was better at adversarial search.

- Indeed, Deep Blue has enormous computational power. It can consider 100–200 billion positions per second and has around 4,000 positions in its opening book.
- In 2006, another world champion, Vladimir Kramnik, was defeated by a machine. This time, it was by Deep Fritz, a German chess computer program.

(2) **Checkers**

- Checkers is another two-player game that can use adversarial search. A computer program called "Chinook" was developed specifically to play in the World Checkers Championship. And in 1990, it reached its goal. Chinook was the first computer program that won the right to play for the World Checkers Championship.
- In contrast to Deep Blue and Deep Fritz, however, Chinook's knowledge of the game was not learned via AI, as its developers programmed everything. Still, we can't discount the fact that it is a powerful application. It has a search space of 5×1020 or 500,000,000,000,000,000,000 sets of possible moves (that's 500 quintillion, in case you're wondering).

(3) "Go"

- AlphaGo is a computer program designed to learn and master the 3,000-year-old board game "Go." It uses machine learning (ML) and deep neural networks and has indeed mastered the game, beating "Go" champions such as Lee Se-dol and Fan Hui.
- Se-dol retired in 2019, declaring that "Even if I become number one, there is an entity that cannot be defeated." He was referring to AI-powered "Go" opponents such as AlphaGo, which has an enormous search space.
- Two-player games such as chess, checkers, and "Go" have come a long way with the help of adversarial search methods and other technologies. Although the same game rules are still used, it's awe-inspiring to think that intelligent machines can also play against humans. People can hone their skills in these types of games by playing against machines.

3.26.4 Types of Algorithms in Adversarial Search

- In a normal search, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an adversarial search, the result depends on the players which will decide the result of the game.
- It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.
- There are following types of adversarial search:
  - Minmax Algorithm
  - Alpha-beta Pruning

3.27 GAME TREE

GQ. Explain game tree.

- A **game tree** is a type of **recursive search function** that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move. They are very useful for artificial intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play.
- The most commonly-cited example is chess, but they are applicable to many situations. Game trees are generally used in board games to determine the best possible move. For the purpose of this article, **Tic-Tac-Toe** will be used as an example.
- The idea is to start at the current board position, and check all the possible moves the computer can make. Then, from each of those possible moves, to look at what moves the opponent may make. Then to look back at the computers. Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion. It will do this for every possible outcome, effectively playing thousands (often more) of games. From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.

GQ. How AI technique is used to solve tic-tac-toe problem?

Heuristics function for Tic-Tac-Toe problem

The board used to play the Tic-Tac-Toe game consists of 9 cells laid out in the form of a 3×3 matrix.

The game is played by 2 players and either of them can start. Each of the two players is assigned a unique symbol (generally 0 and X). Each player alternately gets a turn to make a move. Making a move is compulsory and cannot be deferred. In each move a player places the symbol assigned to him/her in a blank cell.

Seven classes of moves have been designed using the available heuristics. Each class of moves represents a set of functionally cohesive moves in order to achieve a certain objective during a game. These classes of moves are defined and their roles in playing the game are discussed below :

1. **Prioritized selection (PS) :** The PS class of moves selects the blank cell with the maximum priority. If there exist more than one cell with the maximum priority, then any one of them can be selected. The PS class of moves makes sure that the player has control over the most important cells on the board.
2. **Motion (M) :** The M class of moves finds all tracks with only one cell filled with the symbol assigned to the player and other two cells blank. Then one of these tracks with the highest priority is chosen. After that the blank cell with higher priority in the chosen track is selected. The M class of moves makes sure that the player continues filling a track in which there is still a chance to win.
3. **Definitive offense (DO) :** This class of moves finds a track with exactly two cells filled with the symbol assigned to the player and the third cell blank. This blank cell is selected. This moves is meant to provide an immediate win to the player.
4. **Definitive defense (DD) :** This class of moves finds a track with exactly two cells filled with the symbol assigned to the opponent player and the third cell blank. This blank cell is selected. It is meant to prevent the player from an immediate loss. If the

moves are not used then the opponent definitely gets a chance to win in the subsequent move.

5. **Tentative offense (TO) :** This class of moves finds all pairs of intersecting tracks in which both the tracks have exactly one cell filled with the symbol assigned to the player and the other two cells including the common one blank. All such common cells of the intersecting tracks are identified and the one with the maximum priority is selected. This move tries to keep the foundations of victory simultaneously on two tracks. If the TO class of moves can be applied then the player can win in the next move.

6. **Tentative defense (TD) :** This class of moves finds all pairs of intersecting tracks in which both the tracks have exactly one cell filled with the symbol assigned to the opponent player and the other two cells including the common one blank. All such common cells of the intersecting tracks are identified and the one with the maximum priority is selected. It tries to undo the effect of the tentative offense class of moves applied by the opponent. If the move is not applied then the player can lose in the next move.

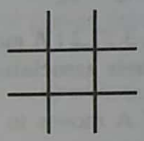
7. **Diagonal correction (DC) :** The above six classes of moves may be insufficient to prevent a loss for the player moving second if either of the two diagonal tracks are filled in the first three moves. This may happen even if the losing player controls the more important cells. It is used to save the player from losing in such conditions.

In Tic-Tac-Toe, player alternate putting marks in a 3×3 array, one mark (*X*) and other mark (0).

Let the evaluation function, e (P) of a position P be given simply by, If *P* is not a wining position for either player.

e (P) = (number of complete row, column, or diagonal that are still open for X)

– (number of complete row, column, or diagonal that are still open for 0)



If P is a win for X,

e (P) = ∞ (a very large value)

If P is a win for 0,

e (P) = – ∞

Thus if *P* is

We have e(P) = 6.4 = 2

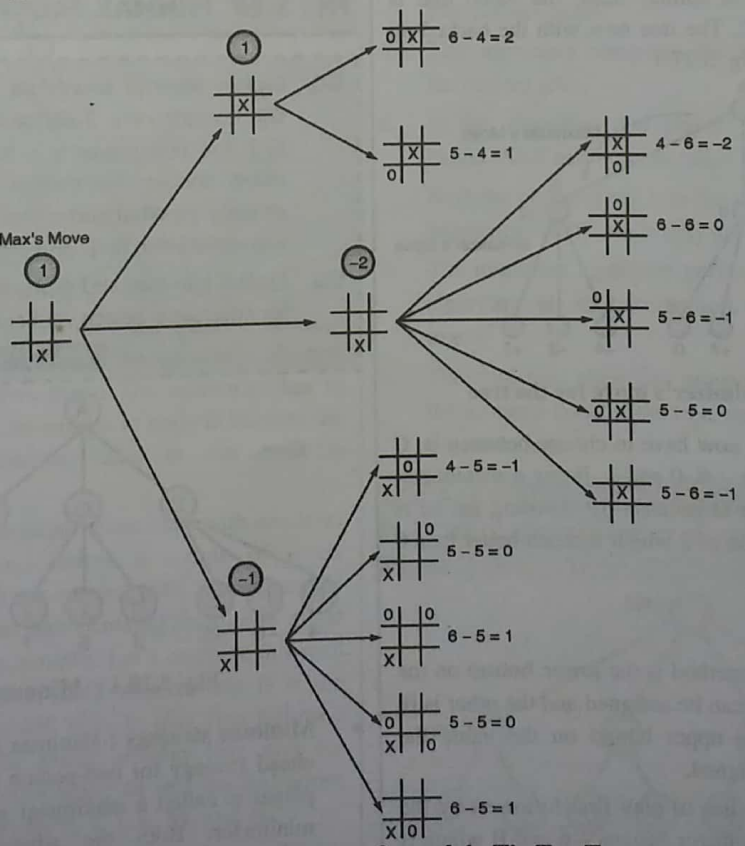


Fig. 3.27.1 : First state of search in Tic-Tac-Toe

3.27.1 tic-tac-toe Problem

GQ. How AI technique is used to solve tic-tac-toe problem ?

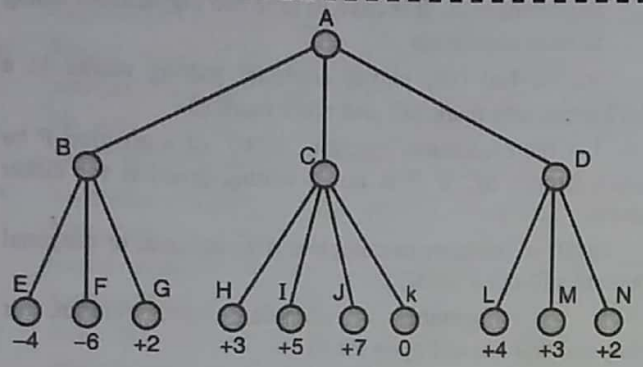


Fig. 3.27.2 : A game tree expanded by two levels and their associated static evaluation function value

8. If A moves to C, then the minimizer will move to K (static evaluation function value = 0) which is the minimum of 3, + 5, 7 and 0. So the value of 0 is backed up at C. On similar lines, the value that is backed up at D is 2. The tree now with the backed up values is given in Fig. 3.27.3.

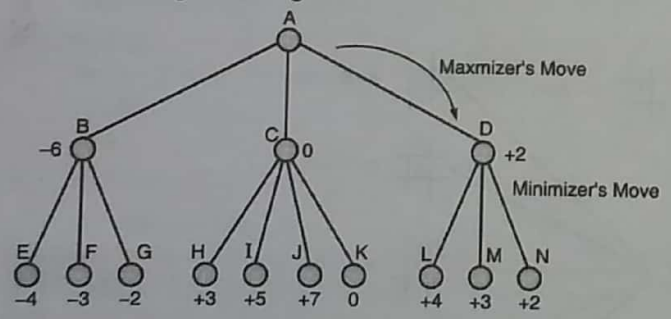


Fig. 3.27.3 : Maximizer's move for the tree

9. The maximizer will now have to choose between B, C or D with the values – 6, 0 and 2. Being a maximizer, he will choose node D because by choosing so, he is sure of getting a value of 2 which is much better than 0 and – 6.

α – β pruning

1. In α – β pruning, α method is the **lower bound** on the value the maximizer can be assigned and the other is β, which represents the upper bound on the value the minimizer can be assigned.
2. Here, the maximizer has to play first followed by the minimizer. Thus maximizer assigns – 6 and B which is passed back to A. This is replaced by 3, value passed by C as A has the maximizer move. Now A will not be examining D and its children. Since value of K is zero and D is having minimizer move making its value 0 or less than 0 only. Thus the tree with the root K will be pruned, which would save a lot of time in searching.

3.27.2 Limitations of Game Trees

1. As mentioned above, game trees are rarely used in real-time scenarios (when the computer isn't given very much time to think).
2. The method requires a lot of processing by the computer, and that takes time.
3. For the above reason (and others) they work best in turn-based games.
4. They require complete knowledge of how to move.
5. Games with uncertainty generally do not mix well with game trees.
6. They are ineffective at accurately ascertaining the best choices in scenarios with many possible choices

3.28 MINMAX PROCEDURE

GQ. Explain Minmax procedure with suitable example.
OR Consider the 2-ply search as shown below : (i) If the first player is a maximizing player, what move should be chosen under the mini-max strategy. (ii) What nodes should not be needed to be examined using α–β pruning technique?

UQ. Explain Min max and Alpha beta pruning algorithms for adversarial search with example.

(MU - Q. 5(b), May 17, 10 Marks)

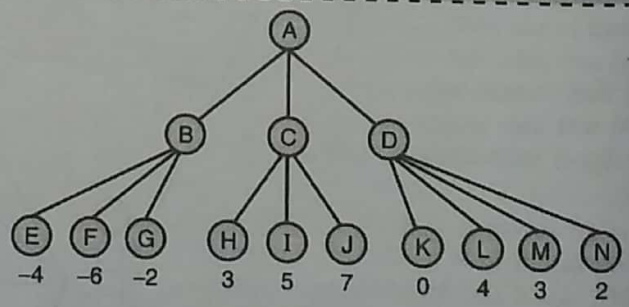


Fig. 3.28.1 : Minmax strategy

- **Minimax strategy :** Minimax strategy is a simple look ahead strategy for two-person game-playing. Here one player is called a maximizer and the other is called a minimizer. Both the adversaries, maximizer and minimizer fight it out to see to that the opponent gets

the minimum benefit while they get the maximum benefit. The plausible move generator generates the necessary states for further evaluation and the static evaluation function "ranks" each of the positions.

Mimax Strategy Algorithm

1. The working of the algorithm is described with the aid of Fig. 3.28.2. Let A be the initial state of the game. The plausible move generator generates three children for that move and the static evaluation function generator assigns the values given along with each of the states.
2. It is assumed that the static evaluation function generator returns a value from – 20 to + 20, wherein a value of – 20 indicates a win for the maximizer and a value of – 20 a win for the minimizer. A value of 0 indicates a tie or draw.
3. It is also assumed that the maximizer makes the first move. (It is not essential so. Even a minimizer can make the first move.) The maximizer, always tries to go to a position where the static evaluation function value is the maximum positive value.

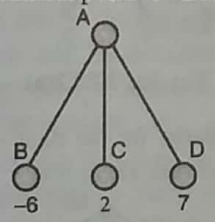


Fig. 3.28.2 : Initial state of the game

4. The maximize being the player to make the first move; will move to node D because the static evaluation function value for that node is the maximum (figure). The same figure shows that if the minimizer has to make the first move, he will go to node B because the static evaluation function value at that node is advantageous to him.
5. But a game-playing strategy never stops with one level but looks ahead, i.e., moves a couple of levels downwards to choose the optimal path. Sometimes, by expanding these nodes and scanning them, one might be forced to retract or rethinks. Let's examine this with the help of Fig. 3.28.2. Let's assume that it is the maximizer again who will have to play first followed by the minimizer. The search strategy here tries for only two moves, the root being M and the leaf nodes being **A, B, C, D, E, F, G, H, I, J, L, M, and N.**
6. Before the maximizer moves to B, C or D it will have to think which move would be highly beneficial to him. In order to evaluate, the children of the intermediate nodes B, C and D are generated and the static evaluation function value generator has assigned values for all the leaf nodes.
7. If A moves to B it is the minimizer who will have to play next. The minimizer always tries to give the minimum benefit to the other and hence he will move to G (static evaluation function value – 6). This value is backed up at N.

3.28.1 Properties of Min Max Algorithm

1. Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory.
2. It provides an optimal move the for player assuming that opponent is also playing optimally.
3. Mini-max algorithm uses recursion to search through the game-tree.
4. Mini-max algorithm is mostly used for game playing in AI. Such as chess, checkers, tic-tac-toe, go and various tow-players game.
5. This algorithm computes the mini-max decisions for the current state.
6. In this algorithm two players play the game, one is called MAX and other is called MIN.

   Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
7. The mini-max algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
8. The mini-max algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

UEx. 3.28.1 : (MU-Dec. 15, Dec. 19 10 Marks)

Problems on min-max search on game tree as shown in the Fig. Ex. 3.28.1.

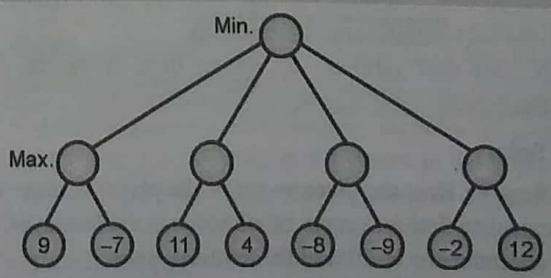


Fig. Ex. 3.28.1

Module 3

Soln. :

Step (I) : Since the given move of the player is max; we calculate first maximum of all nodes.

We begin with left node of the layer above the terminal; to calculate the utility of the left node.

Now, max $\{9, -7\} = 9$

$\therefore$ Utility of the left most node $= 9$ ...(i)

The utility of the next node of the same layer is max $\{11, 4\} = 11$ ...(ii)

Again the utility of the node (beginning from left)

$= \max \{-8, -9\} = -8$

and the utility of the last node of the same layer is max $\{-2, 12\} = 12$

Step II :

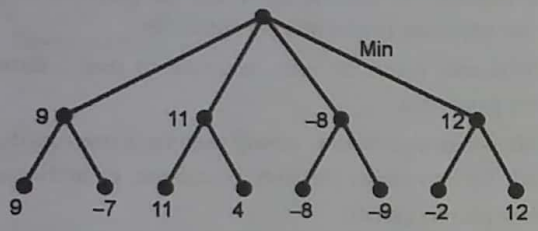


Fig. Ex. 3.28.1(a)

Now, we calculate the utility values, considering one layer at a time, till we reach the root of the tree, i.e. the top-most point.

Here we have 3 layers, so we can directly evaluate min $\{9, 11, -8, 12\} = -8$

Step III : Thus, the best opening move for min is the third node.

This move is called the min max decision. It maximises the utility with the knowledge that the opponent is also playing optimally to minimise it.

Step IV : Thus,

Min-max decision $= \min \{\max (9, -7), \max \{11, 4\},$
$\max \{-8, -9\}, \max \{-2, 12\}\}$
$= \min \{9, 11, -8, 12\} = -8$

UEx. 3.28.2 : (MU - May 18, 10 Marks)

Apply min-max search on game tree given in the Fig. Ex. 3.28.2.

Soln. :

Step I : Here the given move of the player is max; we calculate first maximum of all nodes in the last layer, to determine the utilities of the terminal nodes.

We begin with the left node of the layer.

Again move of the layer is maximum, we choose the maximum of all the utilities.

(i) $\max \{4, 3, 1\} = 4$

$\therefore$ Utility of the left most-node is 4

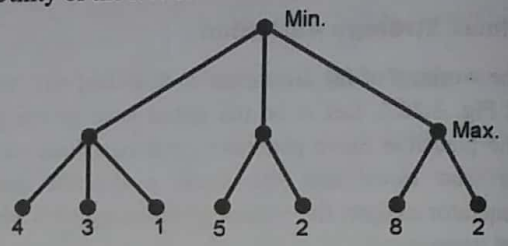


Fig. Ex. 3.28.2

Step II : Again we evaluate maximum of the middle node in the same layer, i.e.

$\max \{5, 2\} = 5$

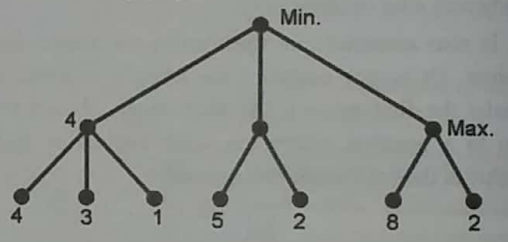


Fig. Ex. 3.28.2(a)

Step III : Similarly, for the right –most of the same layer, we evaluate $\max \{8, 2\} = 8$

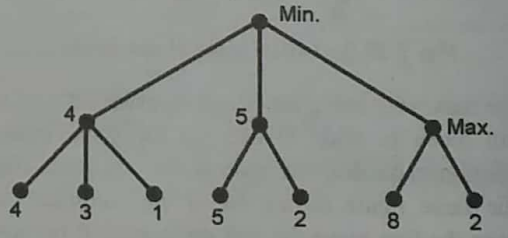


Fig. Ex. 3.28.2(b)

Step IV : Here there are only 3 layers, so we immediately reach to the root. At the root, i.e. the topmost point, min has to choose the minimum value.

So, we evaluate min $\{4, 5, 8\} = 4$

$\therefore$ The best opening move for min is the left most node

Note that this move is called as minmax decision as it maximises the utility under the assumption that the opponent is playing optimally to minimise it.

$\therefore$ Minmax decision $= \min \{\max (4, 3, 1),$
$\max. \{5, 2\}, \max. \{8, 2\}\}$
$= \min \{4, 5, 8\} = 4$

3.29 GAME OF CHANCE

Explain game of chance.

- **Games of chance** are among the oldest of human inventions. The use of a certain type of **animal heel bone** (called the astragalus or colloquially the knucklebone) as a crude die dates to about 3600 BC.
- A game of chance is defined as any game where a **randomizing element** plays a large role in the outcome. This could be as simple as choosing heads or tails, or it might involve rolling dice, picking a card, or moving left instead of right. Skill can still play a large part in such games, although a certain unpredictable variable will always exist.
- It is a game whose outcome is strongly influenced by some randomizing device, and upon which contestants may choose to wager money or anything of monetary value. Common devices used include dice, spinning tops, playing cards, roulette wheels, or numbered balls drawn from a container. A game of chance may have some skill element to it, however, chance generally plays a greater role in determining the outcome than skill. A game of skill, on the other hand, also has an element of chance, but with skill playing a greater role in determining the outcome.
- Games of chance may also involve a certain degree of skill. This is especially true where the player or players have decisions to make based upon previous or incomplete knowledge, such as blackjack. In other games like **roulette and punto banco (baccarat)** the player may only choose the amount of bet and the thing he/she wants to bet on, the rest is up to chance, therefore these games are still considered games of chance with small amount of skills required. The distinction between **'chance'** and **'skill'** is relevant as in some countries chance games are illegal or at least regulated, where skill games are not.

3.30 VARIOUS CATEGORIES OF GAME OF CHANCE

What are the various categories of game of chance?

1. **Arcade games :** While console games are predominant in most nations outside of Japan, arcade games were once the primary form of entertainment for players obsessed with video gaming. From the 1980s to the early '90s, arcade games could be found in most malls and restaurants. The all-time most popular titles include **Pac- Man, Space Invaders, Street Fighter II, and Defender.**
2. **Board games :** The primary component for this category is a board, although dice and playing pieces are also common. Notable examples include monopoly, backgammon, risk, and trivial pursuit. Board games have been discovered dating back to 3500 B.C., which is a testament to their enduring popularity.
3. **Card games :** A 52-card deck is the most common, although special decks can also be purchased near the toy section of any major retail chain. Poker and blackjack are common examples of card games where money is placed on the line (and they also fall into the gambling category), while other examples include magic : The gathering, spades, and old maid.
4. **Computer games :** Instead of being generated by a console such as Nintendo or PlayStation, these games are played directly on a Mac or PC. The quality and complexity of these games range from basic to advanced, with superior computing power giving them an edge over console games. Popular games over the years have included The Sims, Diablo III, Half-Life, and Myst.
5. **Console games :** A video game system generates the images, which are then displayed on a television screen. Console games have enjoyed improved graphics and greater popularity over the last several decades, and the leading companies include Nintendo, PlayStation, and Xbox.
6. **Dice games :** One or more dice are a cornerstone of these games. They have been played for thousands of years, and current favorites include Yahtzee, craps, and poker dice. Since roleplaying and board games frequently use dice, you can expect some products to fall into multiple categories.
7. **Drinking games :** Played in pubs and at keg parties, the objective of these games is often to provide participants with a legitimate excuse to drink more alcohol. While some drinking games emphasize speed or skill, others such as Mexico and Ring of Fire have a more random component.
8. **Gambling games :** This category includes all games where money is wagered by the player in the hopes of receiving a larger payout, whether inside or outside of a casino. Examples include sports betting, keno, the lottery, bingo, and slot machines.

Module 3

9. **Miniature games :** Often associated with miniature wargaming, this category includes games where tiny figures are used to represent opposing armies. Elaborate gaming surfaces often mimic battlefields and towns, while players spend countless hours painting figures. Popular products include warhammer fantasy battle, flames of war, and space marines.
10. **Mobile games :** If a game is available on a mobile phone or tablet computer, then it falls into this category. Thousands of apps are now designed to entertain consumers, with some of the more popular options being Plants vs. Zombies and Fruit Ninja. This category can also apply to handheld video games such as those created for the Game Boy or PlayStation Portable.
11. **Pencil-and-paper games :** If you're looking for a game that only requires pencil and paper, then you've come to the right category. Games of chance under this heading include Battleships and Hangman.
12. **Roleplaying games :** First popularized by Dungeons and Dragons, roleplaying games allow a player to take on the role of a character and engage in adventures in all manner of settings. Much of the game plays out like theatre, while the success of combat and skills are often determined by rolling dice. Notable examples include D and D, Pathfinder, Traveler, Mutants and Masterminds, Call of Cthulhu, and Rifts.
13. **Tile-based games :** Tiles are used as one of the main elements of play, whether they make up the board or comprise the playing pieces. Examples include dominoes, mahjong, and Scrabble.
14. **Travel games :** Also known as "car games" these games are meant to pass the time on long trips. They are often simple and require no playing pieces, although travel editions of many popular board games are also on the market. Playing a travel game is often as simple as counting animals or predicting the color of the next car to be passed on the highway.

Games of chance support the human need for competition, as well as the desire to flirt with uncertainty.

3.31 ALPHA-BETA PRUNING

GQ. Explain Alpha-beta pruning.

UQ. Explain Min max and Alpha beta pruning algorithms for adversarial search with example. (MU - Q. 5(b), May 17, 10 Marks)

UQ. Define Alpha and Beta value in game tree? (MU - Q. 1(c), May 16, 4 Marks)

Alpha-beta search reduces the number of nodes explored in minimax strategy. It reduces the time required for the search.

The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.

Alpha values

1. At a Max node we will store an alpha value
2. A lower bound on the exact minimax score
3. The true value might be $\geq \alpha$
4. If we know Min can choose moves with **score** $< \alpha$
5. Then Min will never choose to let Max go to a node where the score will be α or more.

Beta values

- At a min node
  1. The beta value is an upper bound on the exact minimax score.
  2. The true value might be $\leq \beta$.
  3. If we know Max can choose movers with **score** $> \beta$.
- Then Max will never choose to let Min go to a node where the score will be β or less.

Alpha beta in action

Why can we cut off search?

- $\beta = 2 < \alpha = 3$ where the a value is at an ancestor node.
- At the ancestor node max had a choice to a score of at least 3.
- Max is not go going to move right to let min guarantee a score of 2.

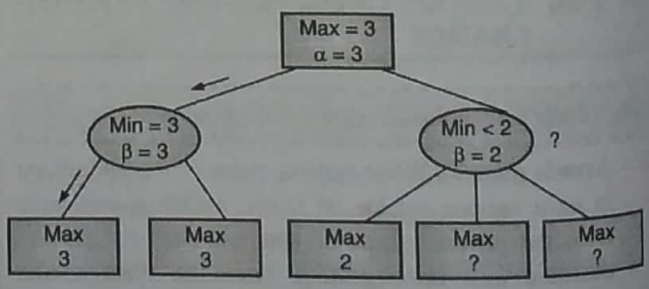


Fig. 3.31.1

- **Search can be discontinued at a node if :**
  1. It is a Max node and the alpha value is ≥ the beta of any min ancestor. This is beta cutoff
  2. Or it is a min node and the beta value is ≤ the alpha of any max ancestor. This is alpha cutoff.

3.31.1 Calculating Alpha-Beta Values

- Alpha-Beta calculations similar to Minimax, but the pruning rule cuts down search.
- Final backed up value of node.
  1. Might be the maximum value.
  2. Or might be an approximation where search cut off.
- Less than the minimax value at a Max node.
- More than the minimax value at a Mid node.
- We don't need to know the true value.

3.31.2 Calculating Alpha Values at a Max Node

- After we obtain the final backed up value of the first child.
  1. We set cx of the node to this value.
- When we get the final backed up value of the second child.
  1. We increase ix if the new value is larger.
- When we have the final child, or if β cutoff.
  1. α value becomes the final backed up value.
  2. Only then can we set the β of the parent Min node

3.31.3 Calculating Beta Values at Min

- After we obtain the final backed up value of the first child, we set β of the node to this value
- When we get the final backed up value of the second child
  1. We decrease β if the new value is smaller
- When we have the final child, or if a – cutoff
  1. β value becomes the final backed up value

3.31.4 Alpha-beta Pruning Example

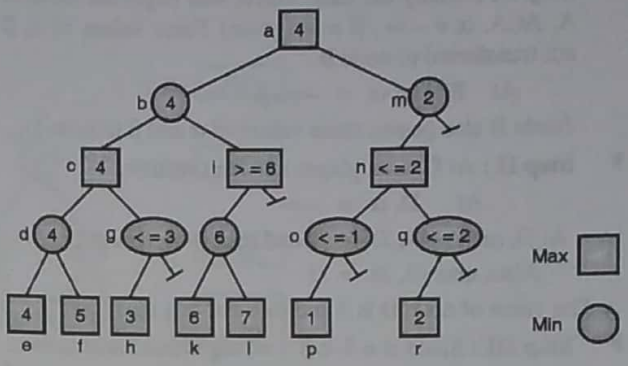


Fig. 3.31.2

- MINIMAX (n, alpha, beta)
- IF n is at the search depth, RETURN V(n)
- FOR each child m of n
- value = MAXIMIN (m, alpha, beta) IF value < beta , beta = value
- IF beta < = alpha, return alpha RETURN beta

Module 3

Performance of Alpha-Beta pruning

1. Efficiency depends on the order in which the nodes are encountered at the search frontier.
2. Optimal - $b^{1/2}$ - if the largest child of a MAX node is generated first, and the smallest child of a MIN node is generated first.
3. Worst - b.
4. Average $b^{3/4}$ - random ordering.

⇒ Example of alpha-beta pruning

Ex. 3.31.1 : Apply alpha-beta pruning on two player search tree shown in the Fig. Ex. 3.31.1.

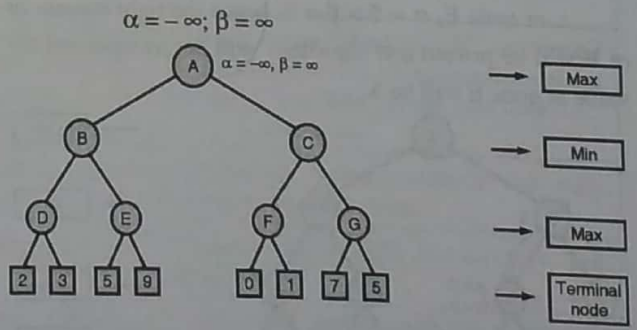


Fig. Ex. 3.31.1: Two player search tree

Soln. :

Step I : Initially the max. Player will begin the move at A. At A, $\alpha = -\infty$, $\beta = \infty$ (given) These values of α, β are transferred to node B.

At B also, $\alpha = -\infty, \beta = \infty$

Node B also passes these values of α and β to node D.

Step II : At D, max player will start its turn.

At D, $\alpha = -\infty$

$\therefore$ At D, max $(-\infty, 2) = 2$ and max $(-\infty, 3) = 3$

Also, max $(2, 3) = 3$

$\therefore$ The value of node D is 3 and value of α is also (say) 3.

Step III : Since $\alpha = 3 < \beta = \infty$; algorithm backtracks to node B. At B, min player will move and value of β $(= \infty)$ will change.

Value of β will be compared to the nodal value of D = 3; and min $(\infty, 3) = 3$

$\therefore$ At B, $\alpha = -\infty$, $\beta = 3$

Step IV : Since $\alpha < \beta$ at B, algorithm traverses the next successor of Node B. Which is node E, and the values of $\alpha = -\infty$, $\beta = 3$ are transferred to node E.

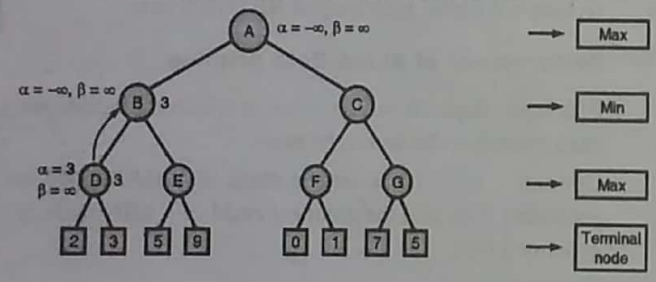


Fig. Ex. 3.31.1(a)

Step V : At E, max player will make a move

$\therefore$ Value of α will change

Now max. $(-\infty, 5) = 5$ or max. $(-\infty, 9) = 9$

$\therefore$ at node E, $\alpha = 5 > \beta = 3$; hence the right success or of E will be pruned and algorithm will not traverse and the value at node E will be 5.

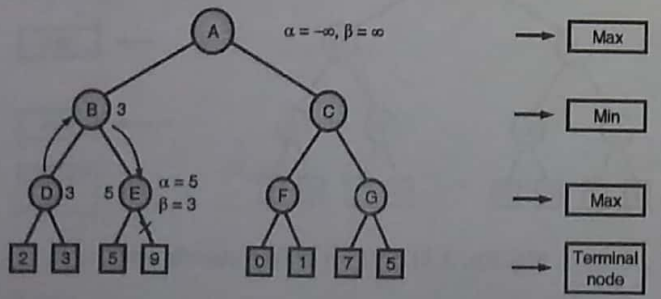


Fig. Ex. 3.31.1(b)

Step VI : Now algorithm again backtrack the tree, from node B to node A.

At A, there is max player, and α will take the maximum value, that is 3, $\because$ max $(\infty, 3) = 3$ and $\beta = \infty$

These two values i.e.; $\alpha = 3$, $\beta = \infty$ will transfer to the successor C. And at node C, again $\alpha = 3$, $\beta = \infty$, and these values again will get transferred to node F.

At F, there is max. Player

Step VII : At node F, we compare the value of α with the left node O, and

max $(3, 0) = 3$

Now node value of F = max $(0, 1) = 1$

$\therefore$ $\alpha = 3 >$ node value of F

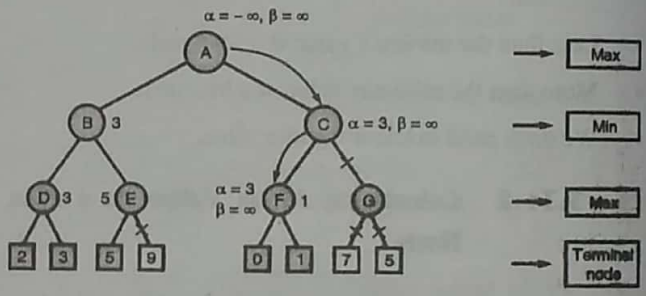


Fig. Ex. 3.31.1(c)

Step VIII : Now the right successor is pruned. ($\because$ $\alpha >$ node value of F)

$\therefore$ node F gets back to node C and at C,

$\alpha = 3$, $\beta = \infty$

Here β will change, since C is min player node. At C node value is 1,

$\therefore$ β = min $(\infty, 1) = 1$

Again $\alpha > \beta$ $(\because \alpha = 3, \beta = 1)$

$\therefore$ Next node G will be pruned and algorithm stops computing at G

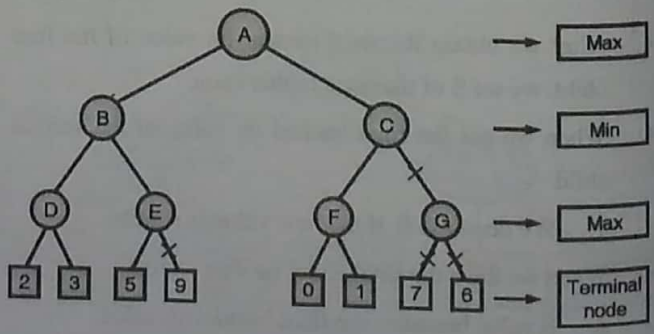


Fig. Ex. 3.31.1(d)

▶ **Step IX :** At C, $\alpha = 3$, $\beta = 1$;

($\because \alpha > \beta$); C goes back to A, and for A, max. (3, 1) = 3

$\therefore$ optimiser value for this optimiser is β.

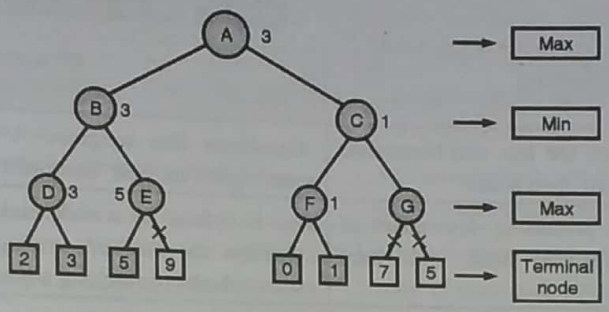


Fig. Ex. 3.31.1(e)

3.32 COMPARISONS IN TABULAR FORM

3.32.1 Difference Between Informed Search & Uninformed Search

UQ. Difference between Informed and Uninformed Search in AI.

(MU - Q. 3(a), Dec. 19, 10 Marks, Q. 4(b), Dec. 18, 10 Marks, Q. 4(b), May 16, 10 Marks)

Module 3

Sr. No.	Parameters	Informed Search	Uninformed Search
1.	Information required in problem solving	Informed strategies use agent's background information about the problem map, costs of actions, approximation of solutions.	Uninformed strategies use only the information available in the problem definition.
2.	Goal state	In informed search, a heuristic is used as a guide that leads to better overall performance in getting to the goal state.	Uninformed search (blind search) has no information about the number of steps or the path costs from the current state to the goal.
3.	Goal state	Instead of exploring the search tree blindly, one node at a time, the nodes that we could go to are ordered according to some evaluation function h(n) that determines which node is probably the "best" to go to next.	They can only distinguish a goal state from a non-goal state. There is no bias to go towards the desired goal.
4.	Search strategies	Also known as "heuristic search", informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal nodes(s).	Also known as "blind search", uniformed search strategies use no information about the likely "direction" of the goal nodes(s).
5.	Efficiency	Using specific knowledge this method gives solution efficiently.	Uninformed search methods are very inefficient in most cases.
6.	Time and memory consumption	Use comparatively much lesser time and memory.	It leads to higher time and memory time complexity.
7.	Examples	Informed search methods : Hill climbing, best-first, greedy search, beam search, A* algorithm AO* algorithm.	Uninformed search methods : Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional.

3.32.2 Differentiate Hill Climbing vs A* Algorithm

UQ. How do you compare Hill climbing technique with A* algorithm.

MU - Q. 2(A), May. 17, 5 Marks, Q. 2(b), May 16,5 Marks

Sr. No.	Hill Climbing	A* Algorithms
1.	Hill climbing algorithms are less deli-beratative, rather than considering all open nodes.	Algorithms like weighted A* systematically explore the search space in 'best' first order.
2.	They expand the most promising descendant of the most recently expanded node until they encounter a solution.	Best is defined by a node ranking function which typically considers the cost of arriving at a node, g, as well as estimated cost of reaching a goal from a node h.
3.	Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.	A* also considers the distance of a node from the goal, d.

3.32.3 Comparison of Hill-Climbing and Simulated Annealing

UQ. Explain Hill Climbing and Simulated Annealing with suitable example.

(MU - Q. 2(a), Dec. 18, 10 Marks, Q. 2(a), May 18, 10 Marks)

	Hill climbing	Simulated Annealing
(1)	Hill climbing gets stuck in a local maxima because downward moves are not allowed.	Simulated Annealing is a variation of hill climbing.
(2)	Hill-climbing can be applied to most combinatorial optimisation problems.	At the beginning of the process, some downhill moves may be made.
(3)	The three algorithms are used to solve the mapping problem, which is the optimal static allocation of communication processes on distributed memory architecture.	The idea is to do enough exploration of the whole space early on so that the final solution is relatively intensive to the starting state.
(4)	A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum	The chances of getting caught of a local maximum are lowered.
(5)	One of the widely discussed example of Hill climbing algorithm is Travelling-salesman problem in which we minimise the distance travelled by the salesman.	Simulated annealing as a computational process is patterned after the physical process of annealing.
(6)	It is a greedy local search as it only looks to its good immediate neighbour state and not beyond that.	Physical substances such as metals are melted (i.e. raised to high energy levels) and then gradually cooled until some solid state is reached.

3.32.4 Difference between Undirectional and Bidirectional Search

Sr. No.	Unidirectional Search Method	Bidirectional Search Method
1.	It uses search tree, start node, goal node as input for starting search.	It has additional information about search trees nodes, along with the start and goal node.
2.	They use only information from the problem definition.	They incorporate additional measures of a potential of a specific state to reach goal.
3.	Sometimes these methods use past exploration.	All these methods use a potential of a state to reach a goal is measured through heuristic functions.
4.	All these techniques are based on pattern of exploration of the nodes in the search tree.	All these techniques totally depend on the evaluated value of each node generated by heuristic function.
5.	In real time problems these search techniques can be costly with respect to time and space.	In real time problmes these 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 uniformed techniques less number of nodes are explored in this case.
7.	**Example :** Breadth Frist Search, Depth First search, Uniform Cost search, Depth Limited search, Iterative Depending DFS.	**Example :** Hill Climbing search, Best First search, A* search, IDA search, SMA* search

Module 3

3.32.5 Difference Between BFS and DFS

UQ. Compare Breadth First Search (BFS), Depth first search (DFS). (MU - Q. 4(a), Dec. 18, 10 Marks)

Parameter	BFS	DFS
Definition	It is an abbreviation for Breadth First Search.	It is an abbreviation for Depth First Search.
Structure of Data	It uses the Queue Data Structure for finding the shortest path.	It uses the Stack Data Structure for finding the shortest path.
Speed	It is comparatively slower than the DFS method.	It is comparatively faster than the BFS method.
Distance from Source	BFS acts better when the target stays closer to the source.	DFS acts better when the target stays farther from the source.
Suitability for Decision Tree	Since BFS considers all of its neighbors, it is not very suitable for the decision trees used in puzzle games.	DFS is comparatively much more suitable for the decision trees. Within a decision, it allows a user to traverse further to augment that decision. If you reach a conclusion, you reach the win situation, and you stop.
Type of List	BFS operates using the FIFO list.	DFS operates using the LIFO list.

Parameter	BFS	DFS
Tracking Method	BFS uses the queue to keep track of the next location that it should visit.	DFS uses the stack to keep track of the next location that it should visit.
Type of Solution	It ensures a shallow path solution. BFS directly finds the shortest path that leads to a solution.	It does not ensure a shallow path solution. DFS first heads to the bottom of any subtree. Then, it backtracks.
Tree Path	It traverses a path according to the tree level.	It traverses a path according to the tree depth.
Backtracking	You don't need to backtrack in BFS.	You need to follow a backtrack in DFS.
Loop Trapping	A user can happen to get trapped in a finite loop.	Any user can possibly get trapped in an infinite loop.
In Case of No Goal	If a user doesn't find a goal, they may have to expand various nodes before they find the solution.	If a user does not find any goal, it may lead to the leaf node backtracking.
Reaching Destination Vertex	You can use BFS to find a single source of the shortest path in the case of an unweighted graph. It is because, in BFS, one can easily reach the vertex with a minimum number of edges from the source vertex.	You might need to traverse through more edges to find and reach the destination vertex from your source.
Suitable Vertices	BFS works better when a user searches for the vertices that stay closer to any given source.	DFS works better when a user can find the solutions away from any given source.
Memory	The amount of memory required for BFS is more than that of DFS.	The amount of memory required for DFS is less than that of BFS.
Complexity of Time	The time complexity of BFS is O (V+E) when a user deploys the Adjacency List and O(V^2) when the user deploys Adjacency Matrix. Here, E refers to the edges, and V refers to the vertices.	The time complexity of BFS is also equivalent to O (V+E) when a user deploys the Adjacency List and O (V^2) when the user deploys Adjacency Matrix, in which E refers to the edges and V refers to the vertices.

Chapter Ends...

❑❑❑