UNIT 3 UNINFORMED & INFORMED SEARCH

Structure **Page No**

3.0 INTRODUCTION

Before an AI problem can be solved it must be represented as a **state space**. In AI, a wide range of problems can be formulated as search problem. The process of searching means a sequence of action that take you from an initial state to a goal state as sown in the following figure 1.

Fig 1: A sequence of action in a search space from initial to goal state.

In the unit 2 we have already examined the concept of a **state space** and adversarial (game playing) search strategy. In many applications there might be multiple agents or persons searching for solutions in the same solution space. In adversarial search, we need a path to take action towards the winning direction and for finding a path we need different type of search algorithms.

Search algorithms are one of the most important areas of Artificial Intelligence. This unit will explain all about the search algorithms in AI which explore the search space to find a solution.

In Artificial Intelligence, Search techniques are **universal problem-solving methods**. Rational agents or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation.

One **disadvantage** of state space representation is that it is not possible to visualize all states for a given problem. Also, the resources of the computer system are limited to handle huge state space representation. But many problems in AI take the form of state-space search.

Many problems in Al take the form of ***state-space search***.

- The ***states*** might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
- The state-space is the configuration of the possible states and how they connect to each other e.g., the legal moves between states.
- When we don't have an algorithm which tells us definitively how to negotiate the state-space we need to search the state-space to find an optimal path from a start slate to a goal state,

We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible. Chess, for example is a very difficult state space search problem.

Searching is the process looking for the solution of a problem through a set of possibilities (state

space).In general, the searching process starts from the initial state (root node) and proceeds by performing the following steps:

- Check whether the current state is the goal state or not?
- Expand the current state to generate the new sets of states.
- Choose one of the new states generated for search depending upon search strategy (for example BFS, DFS etc.).
- Repeat step 1 to 3 until the goal state is reached or there are no more state tube expanded.

Evaluation (properties) of search strategies : A search strategy is characterized by the sequence in which nodes are expanded. Any search algorithms are commonly evaluated according to the following four criteria. The following four essential properties of search algorithms is used to compare the efficiency of any search algorithms.

Completeness: A search algorithm is said to be complete if it guarantees to return a solution, if exist.

Optimality/Admissibility: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

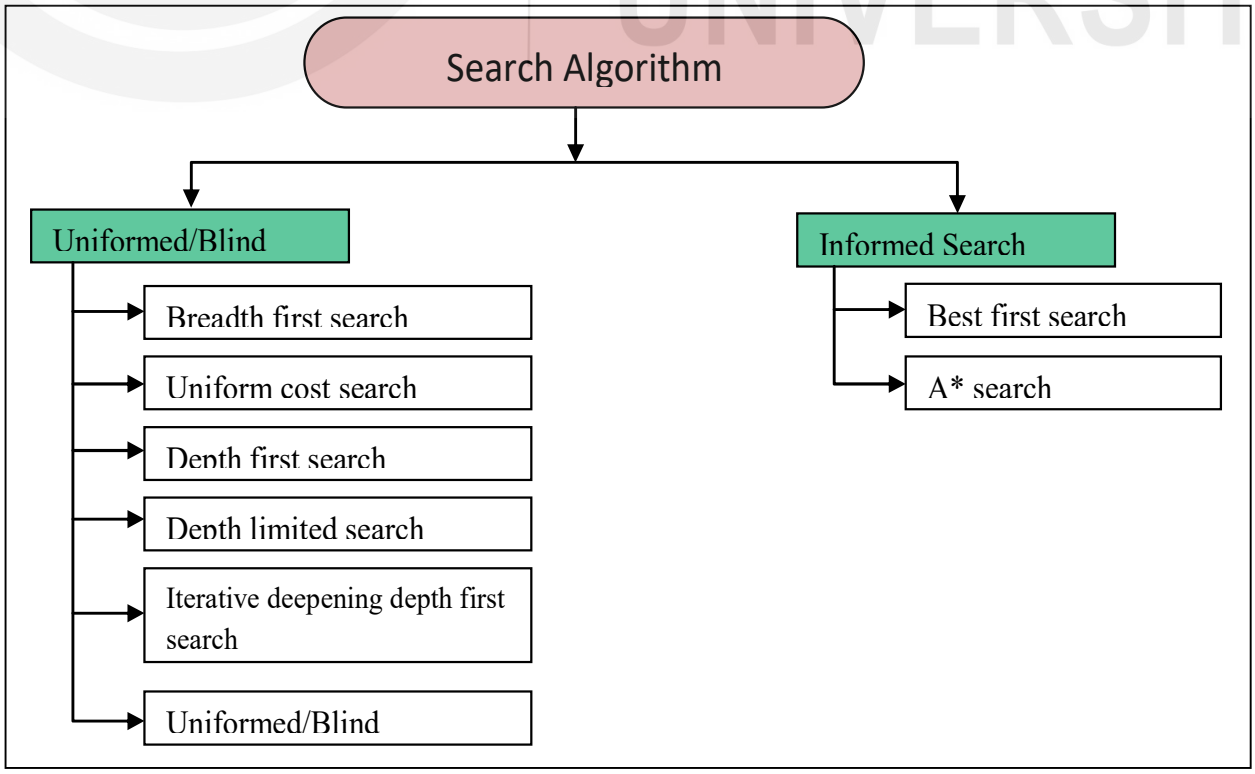
Time Complexity: Time complexity is a measure of time for an algorithm to complete its task. Usually measured in terms of the number of nodes expended during the search.

Space Complexity: It is the maximum storage space required at any point during the search. Usually measured in terms of the maximum number of nodes in memory at a time.

Time and space complexity are measured in terms of:

- *b* - max branching factor of the search tree
- *d* - depth of the least-cost solution
- *m* - max depth of the search tree (may be infinity)

In all search algorithms, the order in which nodes are expended distinguishes them from one another. There are two broad classes of search methods:



Uninformed search is also called Brute force search or Blind search or Exhaustive search. It is called blind search because of the way in which search tree is searched without using any information about the search space. It is called Brute force because it assumes no additional knowledge other than how to traverse the search tree and how to identify the leaf nodes and goal nodes. This search ultimately examines every node in the tree until it finds a goal.

Informed search is also called as Heuristic (or guided) search. These are the search techniques where additional information about the problem is provided in order to guide the search in a specific direction. A heuristic is a method that might not always find the best solution but is guaranteed to find a good solution in reasonable time. By sacrificing completeness, it increases efficiency.

The following table summarizes the differences between uninformed and informed search:

Uninformed Search	**Informed Search**
No information about the path, cost, from the current state to the goal state. It doesn't use domain specific knowledge for searching process.	The path cost from current state to goal state is calculated, to select the minimum cost path as the next state. It uses domain specific knowledge for the searching process.
It finds solution slow as compared to informed	It finds solution more quickly.
Less efficient	More efficient
Cost is high.	Cost is low
No suggestion is given regarding the solution init. Problem to be solved with the given information only.	It provides the direction regarding the solution. Additional information can be added as assumption to solve the problem.
Examples are: ▪ Depth First Search, ▪ Breadth First Search, ▪ Depth limited search, ▪ Iterative Deepening DFS, ▪ Bi-directional search	Examples are: ▪ Best first search ▪ Greedy search ▪ A* search

3.1 OBJECTIVES

After studying this unit, you should be able to:

- Differentiate the Uninformed and informed search algorithm
- Formulate the search problem in the form of state space
- Explain the differences between various uninformed search approaches such as BFS, DFS, IDDFS, Bi-directional search.
- Evaluate the various Uninformed search algorithm with respect to Time, space and Optimality/Admissibility criteria.
- Explain Informed search such as Best-First search and A* algorithm.

- Differentiate between advantages and disadvantages of heuristic search: A* and AO* algorithm
- Differentiate between memory bound search: Iterative Deepening A* and Recursive Best-First Search.

3.2 Formulating search in state space

A state space is a graph, (V, E) where V is a set of nodes and E is a set of arcs, where each arc is directed from one node to another node.

- **V: a node is a data structure that contains** state description, plus, optionally other information related to the parent of the node, operation to generate the node from that parent, and other bookkeeping data.
- **E: Each arc** corresponds to an applicable action/operation. The source and destination nodes are called as parent (**immediate predecessor**) and child **(immediate successor**) nodes with respect to each other. Ancestors(also called predecessors) and descendants (also called successors) node. Each arc has a fixed, non-negative cost associated with it, corresponding to the cost of the action.

Each node has a set of successor nodes. Corresponding to all operators (actions) that can apply at source node's state. Expanding a node is generating successor nodes and adding them (and associated arcs) to the state-space graph. One or more nodes may be designated as start nodes.

A **goal** test predicate is applied to a node to determine if its associated state is a goal state. A **solution** is a sequence of operations that is associated with a path in a state space from a start node to a goal node. The**cost of a solution** is the sum of the arc costs on the solution path.

State-space search is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node.

Hence, initially V={S}, where S is the start node; when S is expanded, its successors are generated, and those nodes are added to V and the associated arcs are added to E. This process continues until a goal node is generated (included in V) and identified (by goal test).

To implement any ***Uninformed search*** algorithm, we always initialize and maintain a list called **OPEN** and put start node of G in OPEN. If after some time, we find **OPEN** is empty and we are not getting "goal node", then terminate with failure. We select a node $\boldsymbol{n}$ from OPEN and if $\boldsymbol{n \in Goal\ node}$, then terminate with success, ***else*** we generate the successor of n (using operator O) and insert them in OPEN. In this way we repeat the process till search is successful or unsuccessful.

Search strategies differ mainly on how to select an OPEN node for expansion at each step of search.

A general search algorithm
1. Initialize: Set = $\{s\}$, where s is a start state.
2.Fail: If $OPEN = \{\ \}$, terminate with failure**.**
3. Select: Select a state, n, from OPEN
4.Terminate: If $n \in Goal\ node$, terminate with success
5. Expand: Generate the successor of n using operator O and insert them in OPEN.
6. LOOP: Goto Step 2

But the problem with the above search algorithm, it is not mentioned that when a node is already visited, then do not revisit that node again. That is "***how we can maintain a part of the state space that is already visited***". So, we have an extension of the same algorithm, where we can save the explicit state space. To save the explicit space, we maintained another list called **CLOSED.**

Thus, to implement any Uninformed search algorithm efficiently, two list **OPEN** and **CLOSED** are used.

Now we can select a node from OPEN and save it in CLOSED. The CLOSED list keeps record of nodes that are Opened. The major difference of this algorithm with the previous algorithm is that when we generate successor node from CLOSED, we check whether it is already in $(OPEN \cup CLOSED)$. If it is already in $(OPEN \cup CLOSED)$, we will not insert in OPEN again, otherwise insert. The following modified algorithm is used to save the explicit space using the list CLOSED.

Modified search algorithm to saving the explicit space

1. Initialize: Set $OPEN = \{s\}, CLOSED = \{\ \}$.
2.Fail: If $OPEN = \{\ \}$, terminate with failure**.**
3. Select: Select a state, n, from OPEN and save n in CLOSED
4.Terminate: If $n \in Goal\ node$, terminate with success
5. Expand: Generate the successor of n using operator O
For each successor, m, insert m in OPEN,only if $m \notin (OPEN \cup CLOSED)$
6. LOOP: Goto Step 2.

Here the OPEN and CLOSED list are used as follows:

OPEN: Nodes are yet to be visited.

CLOSED: Keeps track of all the nodes visited already

Note that initially OPEN list initializes with start state of G (e.g.,$\boldsymbol{OPEN = \{s\}}$**)** and CLOSED list as empty (e.g.,$\boldsymbol{CLOSED = \{\}}$**).**

Insertion or removal of any node in OPEN depends on specific search strategy.

3.2.1 Evaluation of Search Algorithms:

In any search algorithm, we select a node and generate its successor. Search strategies differ mainly on how to select an **OPEN** node for expansion at each step of search. Also, Insertion or deletion of any node from **OPEN** list depends on specific search strategy. Any search algorithms are commonly evaluated according to the following 4 criteria:It is the measure to evaluate the performance of the search algorithms:

- **Completeness:** Guarantees finding a solution whenever one exists.
- **Time Complexity:** How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded.
- **Space Complexity:** How much space is used by the algorithm? Usually measured in terms of the maximum size that the "OPEN" list becomes during the search. The Time and Space complexity are measured in terms of: The branching factor or maximum number of successors of any node and **d**: the depth of shallowest goal node (depth of the least cost solution) and **m:** The maximum depth (length) of any path in the state space (may be infinite).
- **Optimality/Admissibility:** If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Search process constructs a search tree, where **root** is the initial state S, and **leaf nodes** are nodesnot yet been expanded (i.e., they are in OPEN List) or having no successors (i.e., they're dead ends"). Search tree may be infinite because of loops even if statespace is small Search strategies mainly differ on select OPEN**.** Each node represents a partial solution path and cost of the partial solution path) from the start node to the given node. In general, from this node there are many possible paths (and therefore solutions that have this partial path as a prefix.

All search algorithms are distinguished by the order in which nodes are expended. There are two broad classes of search methods: Uninformed search and Heuristic Search. Let us first discuss the Uninformed search.

3.3 UNINFORMED SEARCH

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which searchtree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node. Sometimes we may not get much relevant information to solve a problem.

For Example, suppose we lost our car key, and we are not able to recall where we left, we have to search for the key with some information such as in which places, we used to place it. It may be our pant pocket or may be the table drawer. If it is not there, then we must search the whole house to get it. The best solution would be to search in the places from the table to the wardrobe. Here we need to search blindly with less clue. This type of search is called uninformed search or blind search.

Based on the order in which nodes are expended, we have the following types of uninformed search algorithms:

- Breadth-first search
- Depth-first search
- Uniform cost search
- Iterative deepening depth-first search
- Bidirectional Search

3.3.1 Breadth-first search (BFS):

It is the simplest form of blind search. In this technique the root node is expanded first, then all its successors are expanded and then their successors and so on. **In general, in BFS, all nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.** It means that all immediate children of nodes are explored before any of the children's children are considered. The search tree generated by BFS is shown below in Fig 2.

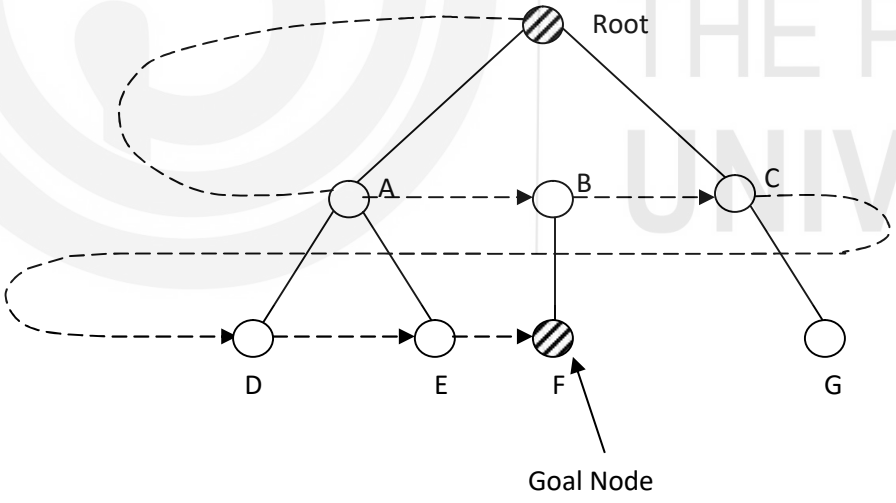


Fig 2 Search tree for BFS

Note that BFS is a brute-search, so it generates all the nodes tor identifying the goal and note that we are using the convention that the alternatives are tried in the left-to-right order.

A BFS algorithm uses a data structure-**queue** that works on FIFO principle. This queue will hold all generated but still unexplored nodes. **Please remember that the order in which nodes are placed on the queue or removal and exploration determines the type of search.**

We can implement it by using two lists called OPEN and CLOSED. The OPEN list contains those states that are to be expanded and CLOSED list keeps track of state already expanded. Here OPEN list is used as a **queue.**

BFS is effective when the search tree has a low branching factor.

Breath-First Search (BFS) Algorithm
1. Initialize: Set = $\{s\}$, where s is a start state.
2.Fail: If $OPEN = \{\ \}$, terminate with failure**.**
3. Select: Remove a left most state (say $\boldsymbol{a}$) from OPEN.
4.Terminate: If $a \in Goal\ node$, terminate with success, else
5. Expend: Generate the successor of node $\boldsymbol{a}$, discard the successors of $\boldsymbol{a}$ if it's already in OPEN, Insert only remaining successors on right end of OPEN [i.e., QUEUE]
6. LOOP: Goto Step 2

Let us take an example to see how this algorithm works.

Example1: Consider the following graph in fig-1 and its corresponding state space tree representation in fig-2. Note that A is a start state and G is a Goal state.

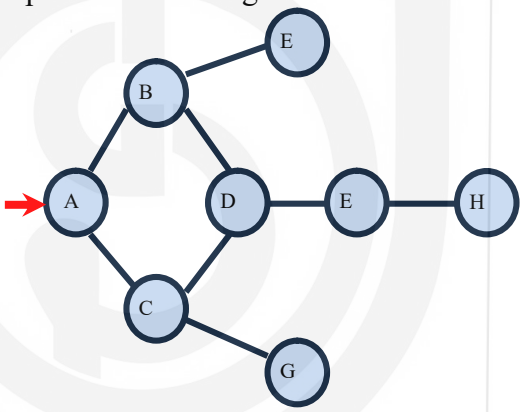


Fig-1 State space graph

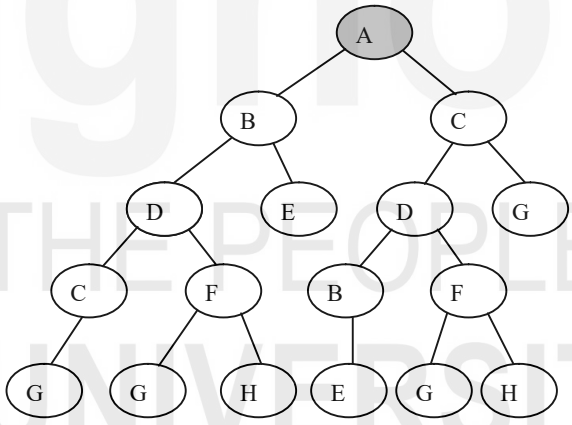
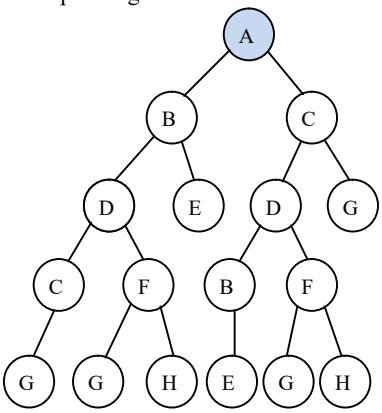


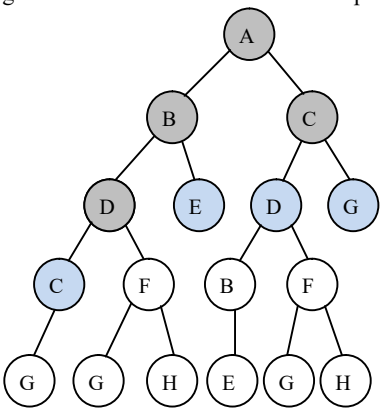
Fig-2 State space tree

Step1: Initially open contains only one node corresponding to the source state A.



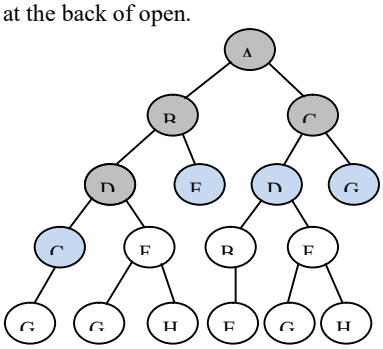
Open= A CLOSED={ }

Step 5: Node D is removed from open. Its children C and F are generated and added to the back of open.



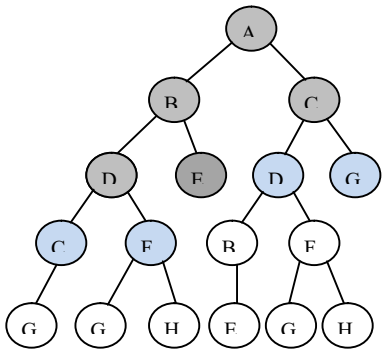
Open= E,D,G,C,F CLOSED=A,B,C,D

Step2: A is removed from open. The node is expended, and its children B and C are generated. They are placed at the back of open.



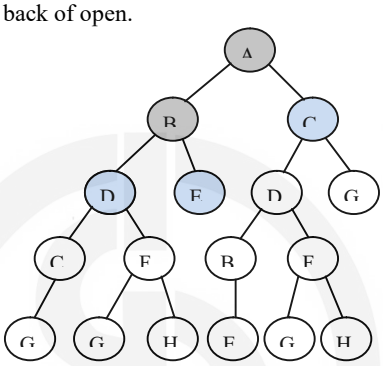
Open= B,C CLOSED={A }

Step 6: Node E is removed from open. It has no children



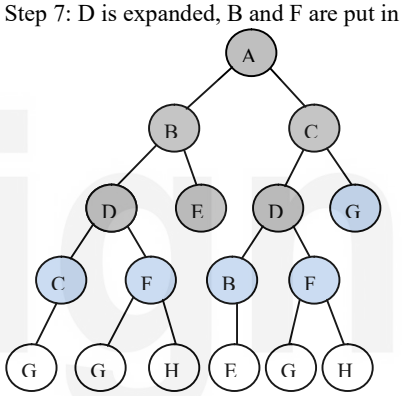
Open= D,G,C,F CLOSED=A,B,C,D,E

Step: 3: Node B is removed from open and is expended. Its children D, E are generated and put at the back of open.



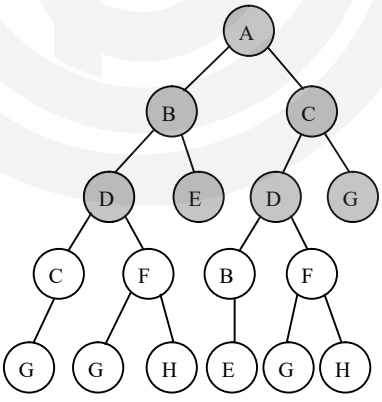
Open= C,D,E
CLOSED={A,B}

Step 7: D is expanded, B and F are put in OPEN.



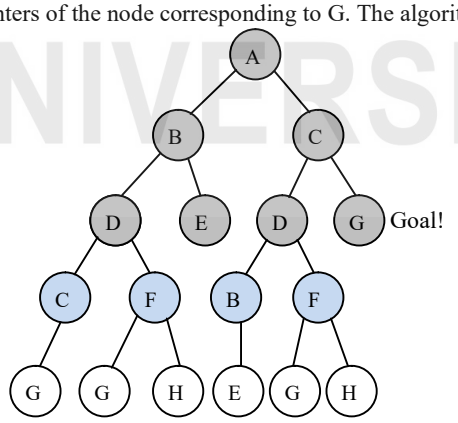
Open= G,C,F,B,F
CLOSED=A,B,C,D,E,D

Step 4: Node C is removed from open and is expanded its children D and G are added to the back of open.



Open=D,E,D,G
CLOSED={A,B,C}

Step 8: G is selected for expansion. It is found to be a goal node. So, the algorithm returns the path ACG by following the parent pointers of the node corresponding to G. The algorithm terminates.



Open=C,F,B,F
CLOSED={A,B,C,D,E.G}

3.3.2 Time and Space complexity of BFS:

Consider a complete search tree of depth d where each non-leaf node has b children (i.e., branching factor), has a total of

$1 + b + b^2 + b^3 + \cdots + b^d = \frac{1.(b^{d+1}-1)}{b-1}$nodes.

Time complexity is the number of nodes generated, so time complexity of BFS algorithm is $O(b^d)$

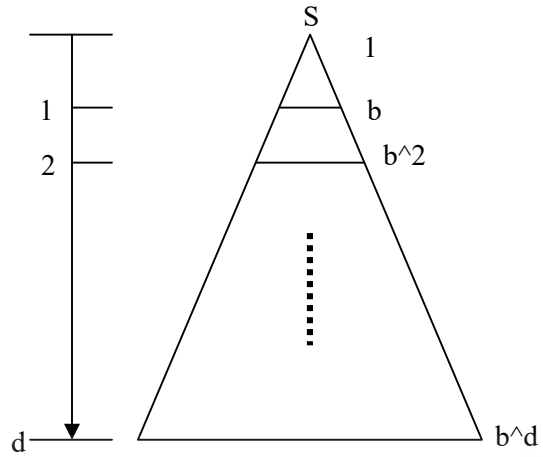


Fig 3 Search tree with b branching factor

For Example, consider a complete search tree of depth 12, where every node at depths $0,1,\ldots,11$ has 10 children (branching factor b=10) and every node at depth 12 has 0 children, then there are

$$1 + 10 + 10^2 + 10^3 + \cdots + 10^{12} = \frac{1.(10^{13} - 1)}{10 - 1}$$

$= \frac{10^{13}-1}{9} = O(10^{12})$ nodes in the complete search tree.

• BFS is suitable for problems with shallow solutions

Space complexity:

BFS has to remember each and every node it has generated. Space complexity (maximum length of OPEN list):

So, space complexity is given by:

$1+b+b^2+b^3+\ldots b^d = O(b^d)$.

Performance of BFS:

- **Time** Required for BFS for tree of b branching factor and d depth is $O(b^d)$.
- S**pace** (memory) requirement for a tree with b branching factor and d depth is also $O(b^d)$
- BFS algorithm is **Complete** (if b is finite).
- BFS algorithm is **Optimal** (if cost = 1 per step)

Space is the bigger problem in BFS as compared to DFS.

3.3.3 Advantages and disadvantages of BFS:

Advantages: BFS has some advantages and are given below

1. BFS will never get trapped exploring bund alley.

2. It is guaranteed to find a solution if one exists.

Disadvantages: BFS has certain disadvantages also. They are given below-

1. Time complexity and Space complexity are both $O(b^d)$ i.e., exponential type. This is very hurdle.

2. All nodes are to be generated in BFS. So, even unwanted nodes are to be remembered (stored in queue) which is of no practical use of the search.

3.3.4 Depth First Search

A Depth-First Search (DFS) explores a path all the way to a leaf before backtracking and exploring another path. That is expand deepest unexpanded node (expand most recently generated deepest node first).

The search tree generated by the DFS is show in figure below:

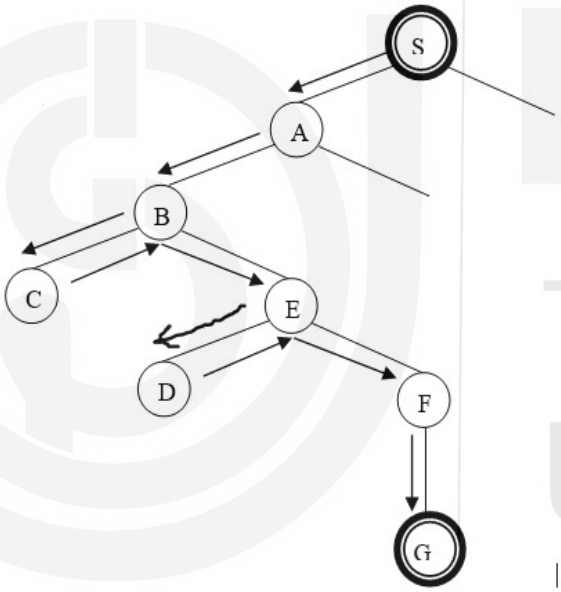


Fig 4 Depth first search (DFS) tree

In depth-first search we go as far down as possible into the search tree/graph before backing up and trying alternatives. It works by always generating a descendent of the most recently expanded node until some depth cut off is reached and then backtracks to next most recently expanded node and generates one of its descendants. So only path of nodes from the initial node to the current node is stored, in order to execute the algorithm. For example, consider the following tree and see how the nodes are expended using DFS algorithm.

Example1:

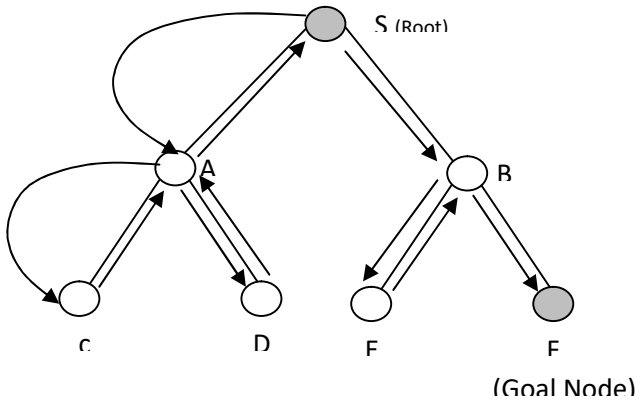


Fig. 5 Search tree for DFS

After searching root node S, then A and C, the search backtracks and tries another path from A. Nodes are explored in the order S, A, C, D, B, E, F.

Here again we use the list OPEN as a STACK to implement DFS. If we found that the first element of OPEN is the Goal state, then the search terminates successfully.

Depth-First Search (DFS) Algorithm

1. Initialize: Set = $\{s\}$, where s is a start state.

2.Fail: If $OPEN = \{\}$, terminate with failure**.**

3. Select: Remove a left most state (say $\boldsymbol{a}$) from OPEN.

4.Terminate: If $a \in Goal\ node$, terminate with success, else

5. Expend: Generate the successor of node $\boldsymbol{a}$, discard the successors of $\boldsymbol{a}$ if it's already in OPEN, Insert remaining successors on **left** end of OPEN [i.e., STACK]

6. LOOP: Goto Step 2

Note: The only difference between BFS and DFS is in **Expend** (step 5). In BFS, we always insert the generated successors at the right end of OPEN list, whereas in DFS at the left end of OPEN list.

Properties of DFS Algorithm:

Supposeb(branching factor), that is Maximum number of successors of any node and M: maximum depth of a leaf node, then

Number of nodes generated (in worst case): $1 + b + b^2 + \cdots + b^m = O(b^m)$

3.3.5 Performance of DFS:

- **Time** Required for DFS for tree of b branching factor and m depth (of shallowest goal node) is $O(b^m)$.
- **Space** (memory) requirement for a tree with b branching factor and m depth (of shallowest goal node) is also $O(bm)$
- BFS algorithm is **Complete** (if b is finite).
- BFS algorithm isnot**Optimal.**

Space is the advantage of DFS as compared to BFS.

3.3.6 Advantages and disadvantages of DFS:

Advantages:

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
- The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space.

Example1: Consider the following graph in fig-1 and its corresponding state space tree representation in fig-2. Note that A is a start state and G is a Goal state.

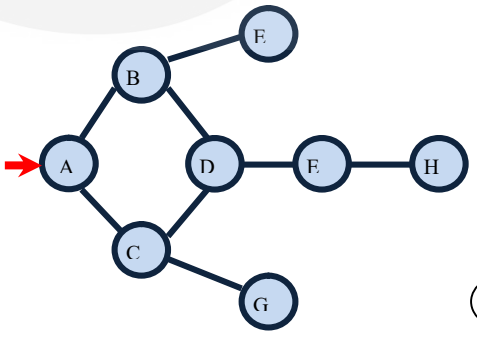


Fig-1 State space graph

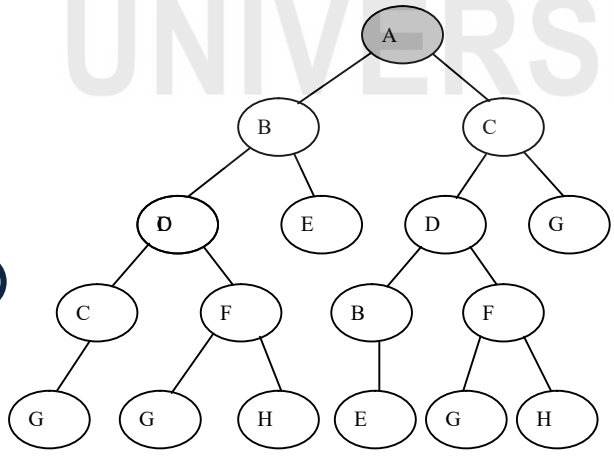
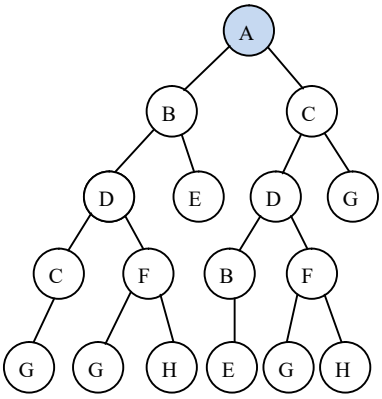


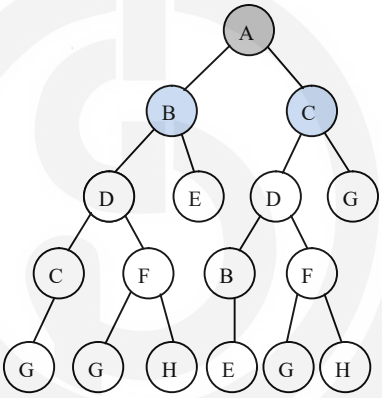
Fig-2 State space tree

Step 1: Initially open contains only one node corresponding to the source state A.



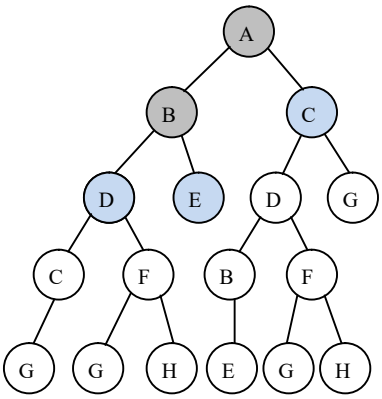
Open= A CLOSED={ }

Step 2: A is removed from open. The node A is expanded, and its children B and C are generated. They are placed at the from of open.



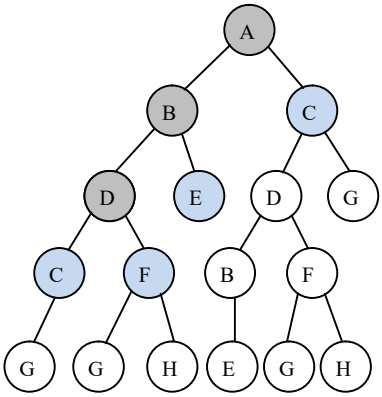
Open= B,C CLOSED={A}

Step 3: Node B is removed from open and is expended. Its children D, E are generated and put at the front open.



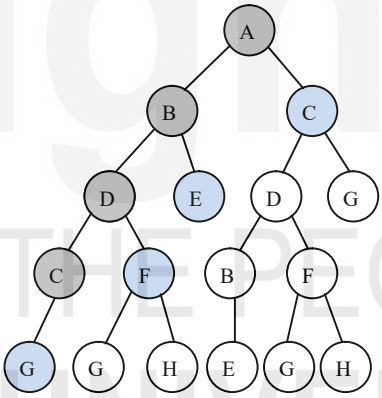
Open= D,E,C CLOSED={A,B}

Step 4: Node D is removed from open and is expanded. Its children C and F are added to the front of open.



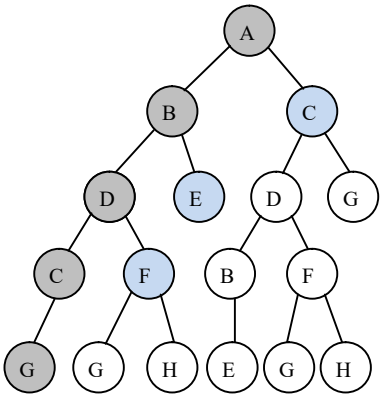
Open= C,F,E,C CLOSED=A,B,D

Step 5: Node C is removed from open. Its children G and F is added to the front of open.



Open= G,F,E,C CLOSED=A,B,D,C

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



Open= F,E,C CLOSED=A,B,D,C,G

3.3.7Comparison of BFS and DFS

BFS goes level wise , but requires more space as compared to DFS. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is $O(b^d)$.

DFS: The problem with this appraoch is, if there is a node close to root, but not in first few subtrees explored bt DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges.)

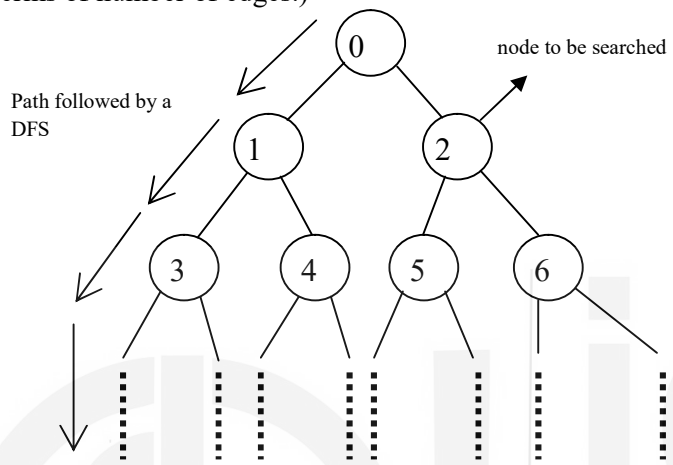


Fig 6: Path sequence in DFS

DFS: The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).

Suppose, we want to find node- '2' of the given infinite undirected graph/tree. A DFS starting from node-0 will dive left, towards node 1 and so on.

Whereas, the node 2 is just adjacent to node 1.

Hense, a DFS wastes a lot of time in coming back to node 2.

An **Iteratie Deepenign Depth First Search** overcomes this andquickly find the required node.

3.4 Iterative Deepening Depth First Search (IDDFS)

Iterative Deepening Depth First Search (IDDFS) neither suffers the drawbacks of BFS nor DFS on trees. It takes the advantages of both the strategy.
It begins by performing DFS to a depth of zero, then depth of one, depth of two, and so on until a solution is found or some maximum depth is reached.
It is like BFS in that it explores a complete layer of new nodes at each iteration before going to next layer.
It is likes DFS for a single iteration.

It is preferred when there is a large search space, and the depth of a solution is notknown. But it performs the wasted computation before reaching the goal depth. Since IDDFS expends all nodes at a given depth before expending any nides at greater depth, it is guaranteed to find a shortest-length (path) solution from initial state to goal state.
At any given time, it is performing a DFS and never searches deeper than depth 'd'. Hence, it uses same space as DFS.
Disadvantage of IDDFS is that it performs wasted computation prior to reaching the goal depth.

Algorithm (IDDFS)

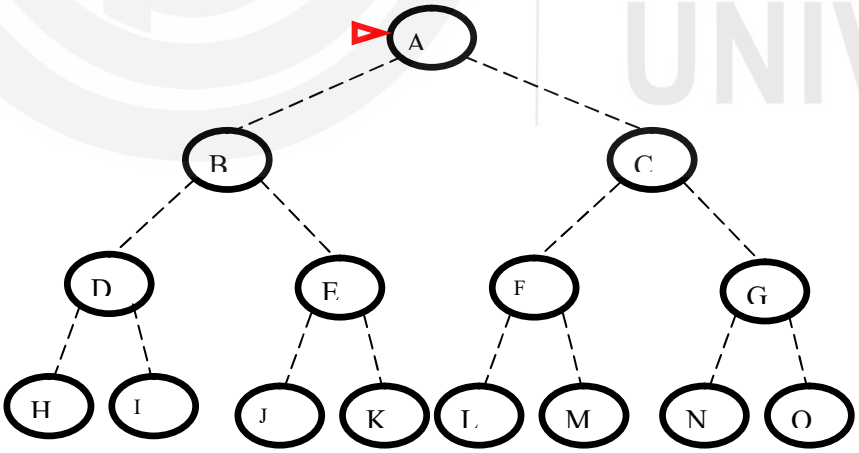
$Initialized\ d = 1/*$ depth of search tree $*/$, found $=$ false
$While\ (Found\ =\ False)$
$DO\{$
$perform\ a\ depth\ first\ search\ from\ start\ to\ depth\ d.$
$if\ goal\ state\ is\ obtained$
$then\ Found\ =\ true$
$else$

$discard\ the\ nodes\ generated\ in\ the\ search\ after\ depth\ d$
$(i.e.\ d+1\ onwards\ till\ last\ of\ tree)$
$\}/*\ end\ while\ */$

$Report\ the\ solution, if\ obtained$
$Stop$

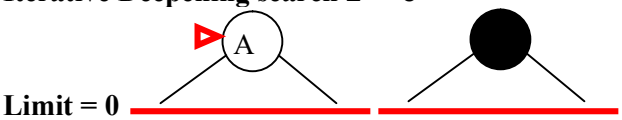
Let us consider the following example to understand the IDDFS:

- Here initial state is A and goal state is M :

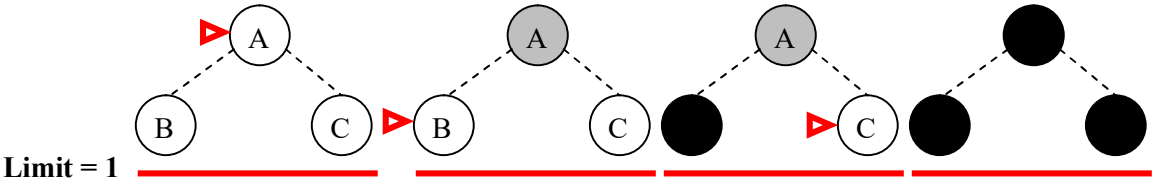


The Iterative deepening search proceeds as follows:

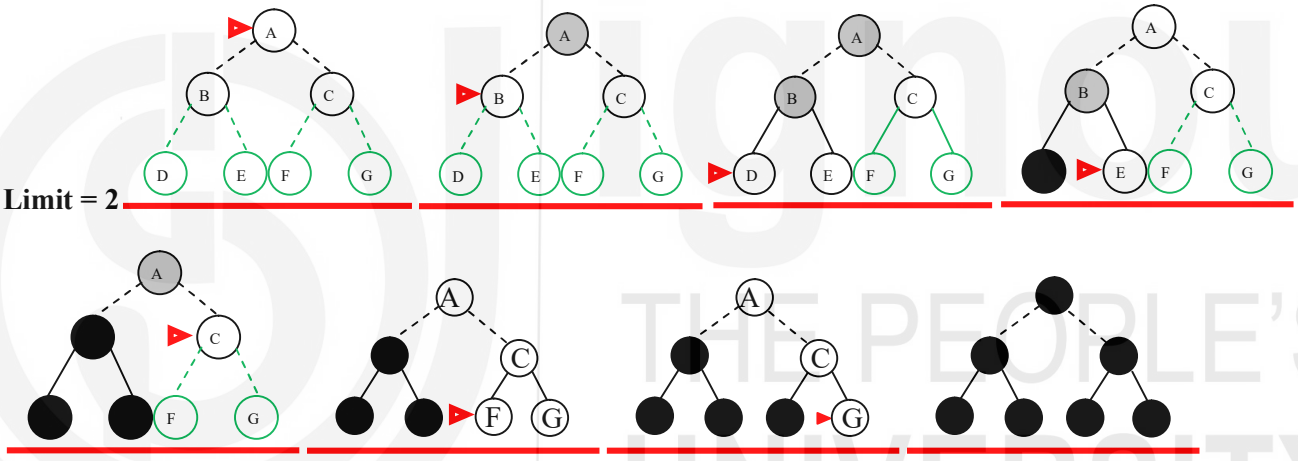
Iterative Deepening search $L = 0$



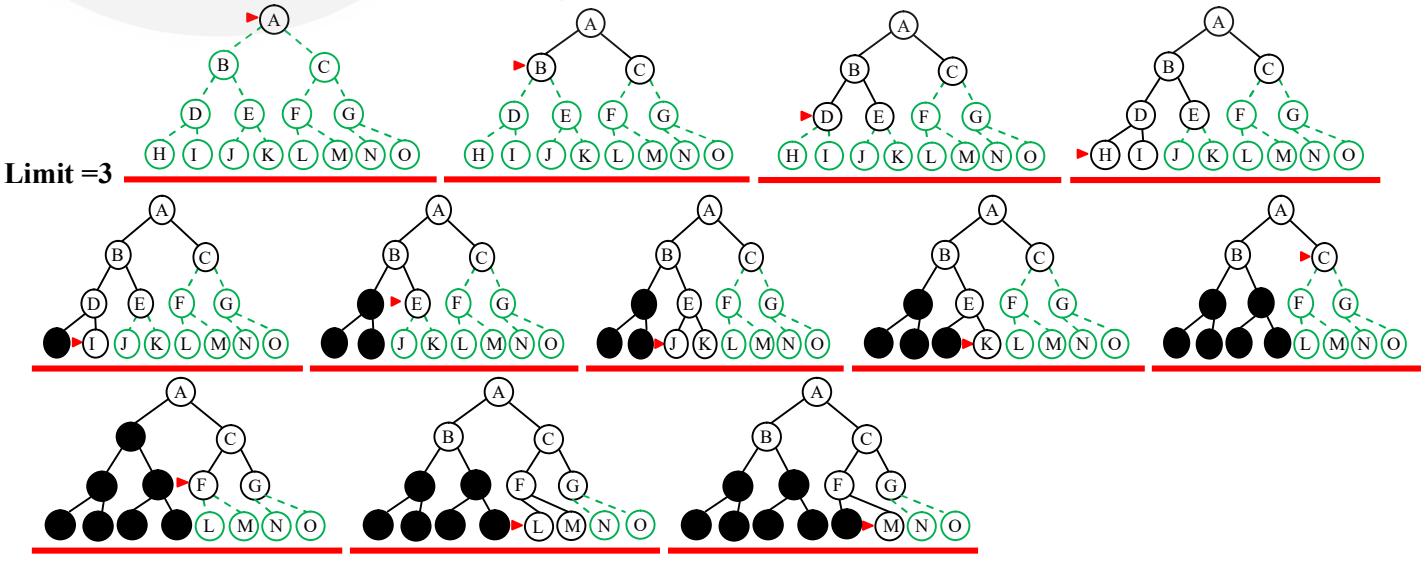
Iterative Deepening search $L = 1$



Iterative Deepening search $L = 2$



Iterative Deepening search $L = 3$



3.4.1 Time and space complexities of IDDFS

The time and space complexities of IDDFSalgorithm is $O(b^d)$ and O(d) respectively.

It can be shown that depth first iterative deepening is **asymptotically optimal,** among brute force tree searches, in terms of time, space and length of the solution. In fact, it is linear in its space complexity like DFS, and is asymptotically optimal to BFS in terms of the number of nodes expanded.

Please note that in general iterative deepening is preferred uninformed search method when there is large search space, and the depth of the solution is unknown. Also note that iterative deepening search is analogous to BFS in that is explores a complete layer going to the next layer.

3.4.2 Advantages and Disadvantages of IDDFS:

Advantages:

1. It combines the benefits of BFS and DFS search algorithms in terms of fast search and memory efficiency.
2. It is guaranteed to find a shortest path solution.
3. It is a preferred uniformed search method when the search space is large and the depth of the solution is not known.

Disadvantages

1. The main drawback of IDDFS is that it repeats all the work from the previous phase. That is, it performs wasted computations before reaching the goal depth.
2. The time complexity is $O(b^d)$ i.e., exponential type only.

The following summarizes when to use which algorithm:

DFS	BFS	IDDFS
Many solutions exist Know (or have a good estimate of) the depth of solution.	Some solutions are known to be shallow	Space is limited and the shortest solution path is required

3.5 Bidirectional Search

This search technique expands nodes from the start and goal state simultaneously. Check at each stage if the nodes of one have been generated by the other. If so, the path concatenation is the solution.

- This search is used when a problem has a single goal state that is given explicitly and all the node generation operators have inverses,

- So, it is used to find shortest path from an initial node to goal node instead of goal itself along with path.
- It works by searching forward from the initial node and backward from the goal node simultaneously, by hoping that two searches meet in the middle.
- Check at each stage if the nodes of one have been generated by the other, i.e., they meet in the middle.
- If so, the path concatenation is the solution.

Thus, the BS Algorithm is applicable when generating predecessors is easy in both forward and backward directions and there exist only 1 or fewer goal states. The following figure illustrate how the Bidirectional search is executed.

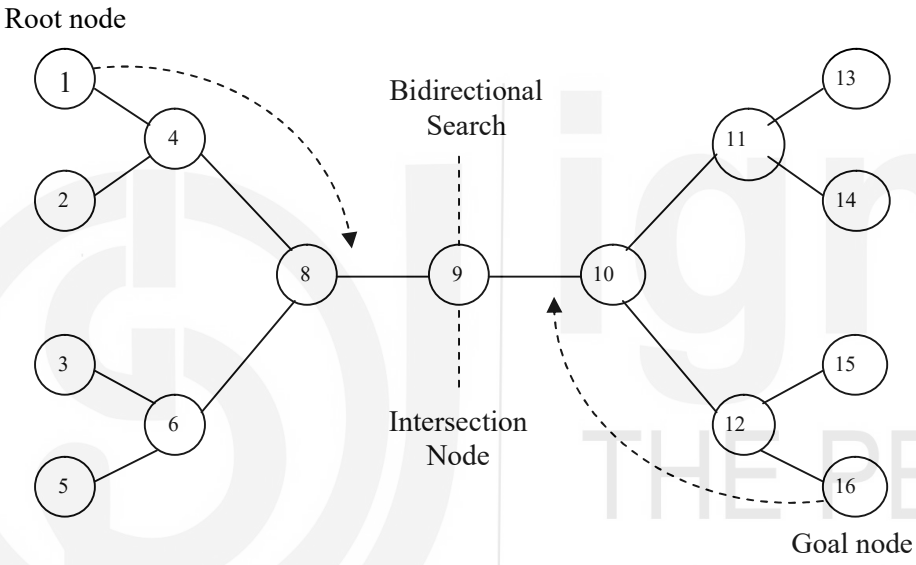


Fig 7 Bidirectional search

We have node 1 as the start/root node and node 16 as the goal node. The algorithm divides the search tree into two sub-trees. So, from start node 1, we do a forward search and at the same time, we do a backward search from goal node 16. The forward search traverse's nodes 1, 4, 8, and 9 whereas the backward search traverses through nodes 16, 12, 10, and 9. We see that both forward and backward search meets at node 9 called the intersection node. So, the total path traced by forwarding search and the path traced by backward search is the optimal solution. This is how the BS Algorithm is implemented.

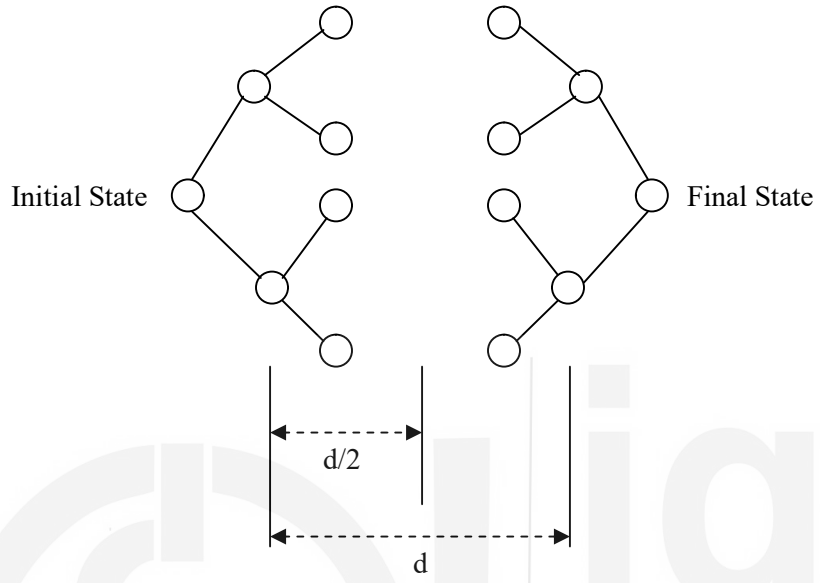
Advantages:

- Since BS uses various techniques like DFS, BFS, Depth limited search (DLS) etc, it is efficient and requires less memory.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Practically inefficient due to additional overhead to perform insertion operation at each point of search.

Time complexity:
The total number of nodes expended in Bidirectional search is= $2b^{d/2} = \boldsymbol{O\left(b^{d/2}\right)}$, where b is a branching factor and d is the depth of the shallowest goal node.



Finally, the Bidirectional search is:

- Complete? Yes
- Time Complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$
- Optimal: Yes (if step cost is uniform in both forward and backward directions)

3.6 Comparison of Uninformed search strategies

The following table-1compare the efficiency of uninformed search algorithms. These are the measure to evaluate the performance of the search algorithms:

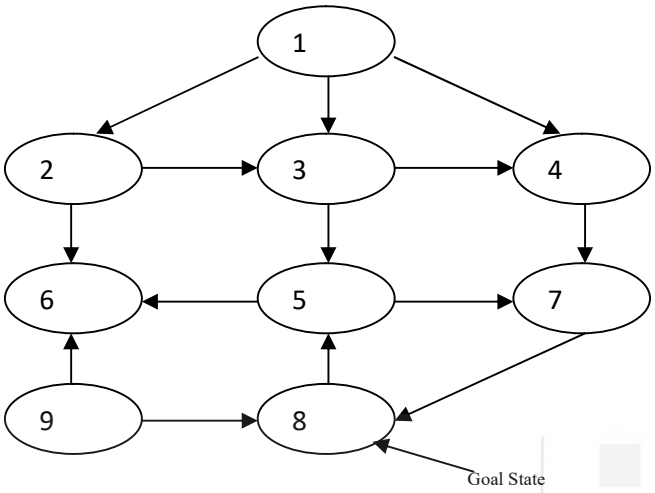
Table -1 Performance of uninformed search algorithm

	BFS	**DFS**	**IDDFS**	**Bidirectional Search (if applicable)**
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

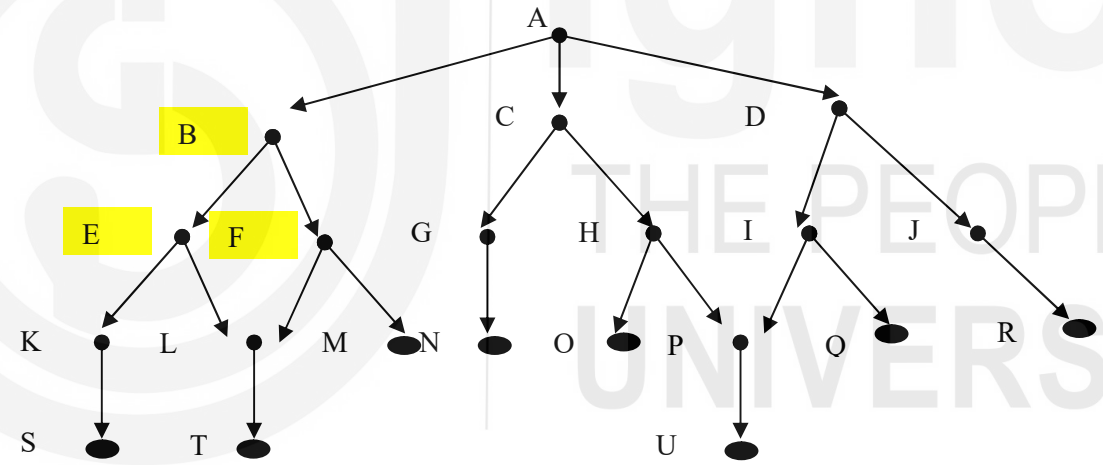
Were
b=branching factor
d=depth of shallowest goal state
m=Maximum depth of the search space

☞ **Check Your Progress 1**

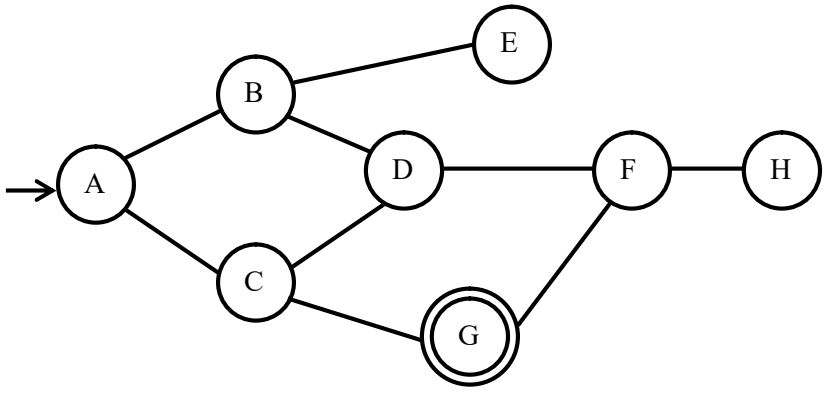
Q.1 Apply BFS and DFS algorithm on the following graph, clearly show the contents of OPEN and CLOSE list.



Q.2 Apply BFS algorithm on the following tree (M is goal node)



Q.3 Apply BFS algorithm on the following graph.



Let A be the state and G be the final or goal state to be searched.

Q.4 Compare the Uninformed search algorithm with respect to Time, space, Optimal and Complete.

3.7 INFORMED (HEURISTIC) SEARCH

Uninformed (blind) search is inefficient in most cases because they do not have any domain specific knowledge about goal state. Heuristic Search Uses domain-dependent (heuristic) information beyond the definition of the problem itself in order to search the space more efficiently.

The following are some ways of using heuristic information:

- Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time:
- Deciding that certain nodes should be discarded, or pruned, from the search space.
-

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently thanan uninformed search strategy. Informed search is also called a **Heuristic search**.
Heuristics is a guess work, or additional information about the problem. It may miss the solution, if wrong heuristics is supplied. However, in almost all problems with correct heuristic information, it provides good solution in reasonable time
Informed search can solve much complex problem which could not be solved in another way.
We have the following informed search algorithm:

1. Best-First Search
2. A* algorithm
3. Iterative Deepening A*

3.7.1Strategies for providing heuristics information:

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

"Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal"

Heuristic Function:

Heuristic information is provided in form of function called **heuristic function.**

- ❖ Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- ❖ It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- ❖ Heuristic function estimates how close a state is to the goal. It is represented byh(n), and it calculates the cost of an optimal path between the pair of states.
- ❖ The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- ❖ This technique always uses to find solution quickly.

- Informed Search Define a heuristic function. h(n). that estimates the "goodness" of a node n.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description of how close we are to a goal.
- Specifically, h(n) = estimated cost (or distance) of minimal cost path from state 'n' to a goal state.

A heuristic function at a node n is an estimate of the optimum cost from the current node toa goal. Denoted by *h(n)*

h(n) = estimated cost of the cheapest path from node n to a goal node

For example, suppose you want to find a shortest path from Kolkata to Guwahati, then heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati, that is

$h(Kolkata) = \boldsymbol{euclideanDistance(Kolkata, Guwahati)}$

3.7.2 Formulation of informed (heuristic) search problem as State Space

Informed search problems can be represented as state space. The state space of a problem includes: an Initial state, one or more goal state, set of state transition operator O (a set of rules), used to change the current state to another state and a heuristic function h.

In general, a state space is represented by 5 tuples as follows: S: $[S, s_0, O, G, h]$

Where S: (Implicitly specified) Set of all possible states (possibly infinite).

s_0: start state of the problem, $s_0 \in S$.

O: Set of state transition operator, each having same cost. This is used to change the state from one state to another. It is the set of arcs (or links) between nodes

G: Set of Goal state, $G \subseteq S$.

$h()$: A heuristic function, estimating the distance toa goal node.

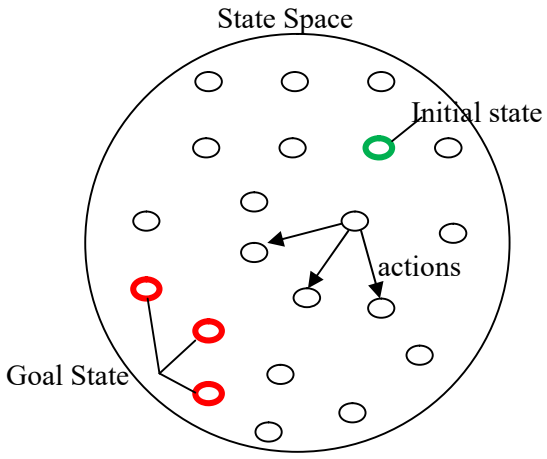


Fig 8 State space with initial and goal node

We need to find a sequence of actions which transform the agent from the initial sate s_0 to Goal state G. State space is commonly defined as a directed graph or as a tree in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.
Thus, the problem is solved by using the rules (operators), in combination with an appropriate control strategy, to move through the problem space until a path from **initial state to a goal state** is found. This process is known as **search**. A solution path is a path in state space from s_0(initial sate) to G (Goal state).

We have already seen an OPEN list is used to implement an uninformed (Blind) search (section 3.2). But the problem with using only one list OPEN is that, it is not possible to keep track of the node which is already visited. That is "***how we can maintain a part of the state space that is already visited***". To save the explicit space, we maintained another list called **CLOSED.** Now we can select a node from OPEN and save it in **CLOSED**. Now, when we generate successor node from CLOSED, we check whether it is already in $(OPEN \cup CLOSED)$. If it is already in $(OPEN \cup CLOSED)$, we will not insert in OPEN again, otherwise insert.

3.7.3 Best-First Search

Best first search uses an evaluation function f(n) that gives an indication of which node to expand next for each node. Every node in a search space has an evaluation function (heuristic function) associated with it. A heuristic function value h(n) on each node indicates how the node is from the goal node. Note that Evaluation function=heuristic cost function (in case of minimization problem) OR objective function(in case of maximization).Decision of which node to be expanded depends onvalue of evaluation function. Evaluation value= cost/distance of current node from goal node and for goal node evaluation function value=0

Based on the evaluation function, f(n), Best-first search can be categorized into the following categories:

1) Greedy Best first search
2) A* search

The following 2 list (**OPEN** and **CLOSED**) are maintained to implement these two algorithms.

1. **OPEN** – all those nodes that have been generated & have has heuristic function applied to them but have not yet been examined.
2. **CLOSED**- contains all nodes that have already been examined.

3.7.4Greedy Best-First search:

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 advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the **priority queue (**or to store the heuristic function value).

Best first search algorithm:
1. Initialize: Set $OPEN = \{s\}$, CLOSED={ };
$f(s) = h(s)$
2.Fail: If $OPEN = \{\}$, terminate with failure**.**
3. Select: Select the minimum cast state $\boldsymbol{a}$ from OPEN, save n in CLOSED**.**
4.Terminate: If $a \in Goal\ node$, terminate with success and return f(n), else
5. Expend: For each successor, m of n,
If $m \notin [OPEN \cup CLOSED]$
$Set\ f(m) = h(m)\ and\ \ Insert\ m\ in\ OPEN$
……
6. LOOP: Goto Step 2

OR Pseudocode (for Best-First Search algorithm)

Step 1: Place the starting node into the OPEN list.
Step 2: If the OPEN list is empty, Stop and return failure.
Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
Step 4: Expand the node n, and generate the successors of node n.
Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
Step 7: Return to Step 2.

Advantages of Best-First search:

- Best first search can switch between BFS and DFS, thus gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

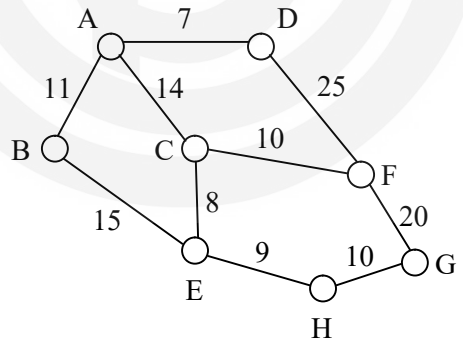
Disadvantages of Best-First search:

- Chances of getting stuck in a loop are higher.
- It can behave as an unguided depth-first search in the worst-case scenario.

Consider the following example for better understanding of greedy Best-First search algorithm.

Example1: Consider the following **example** (graph) with heuristic function value h(n)[Fig 2] which illustrate the greedy Best-first search. Note that in the following example, heuristic function is defined as

$$\boldsymbol{h_{SLD}} = straight\ line\ distance\ from\ n\ to\ goal$$



Let heuristic function value h(n) for each node n to goal node G is defined as

Straight line distance
A→G=h(A)=40
B→G=h(B)=32
C→G=h(C)=25
D→G=h(D)=35
E→G=h(E)=19
F→G=h(F)=17

H→G=h(H)=10
G→G=h(G)=0

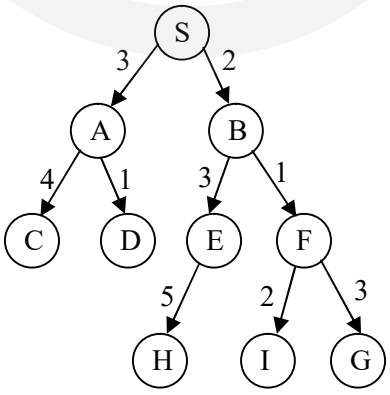
h(n)= straight line distance from node n to G

Note that h(G)=0

The nodes added/deleted from OPEN and CLOSED list using Best-First Search algorithm are shown below.

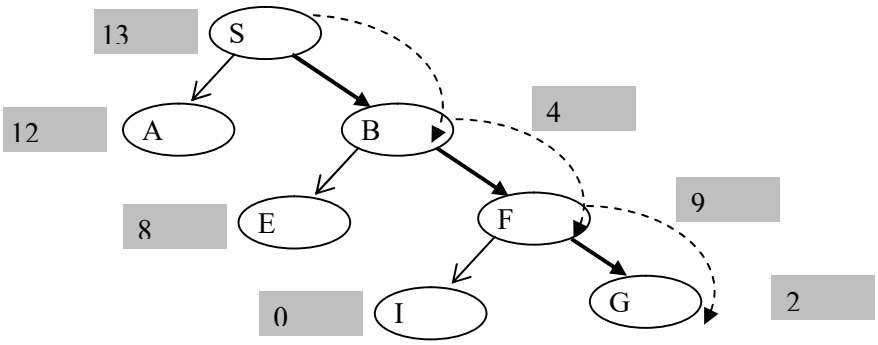
OPEN	**CLOSED**
[A]	[ ]
[C,B,D]	[A]
B,D	A,C
F,E,B,D	A,C
G.E.B.D	A,C,F
E,B,D	A,C,F,G

Explanation are as follows:
Step1:initially OPEN list start with start state 'A' and CLOSED list with empty.
Step2: Children of A={C[25], B[32] D[35]}, so
OPEN={C[25], B[32], D[35]} therefore Best=C, so expend C node next.
Step3: Children of C={E[19],F[17]}, so
OPEN={F[17],E[19], B[32], D[35]} therefore Best=F, so expend node F next.
Step4: Children of F={G[0]}, therefore OPEN= {G[0], E[19], B[32], D[35]} Best=G, this is a goal node so Stop.
Finally, we got the shortest path: A→C→F→G and cost is 44.
Example2: Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	**h(n)**
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13

Here, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]
Iteration2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be **S----> B----->F----> G**

Evaluation of Best-First Search algorithm:

Time Complexity:

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

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.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

Example3: Apply Greedy Best-First Search algorithm on the following graph (L is a goal node).

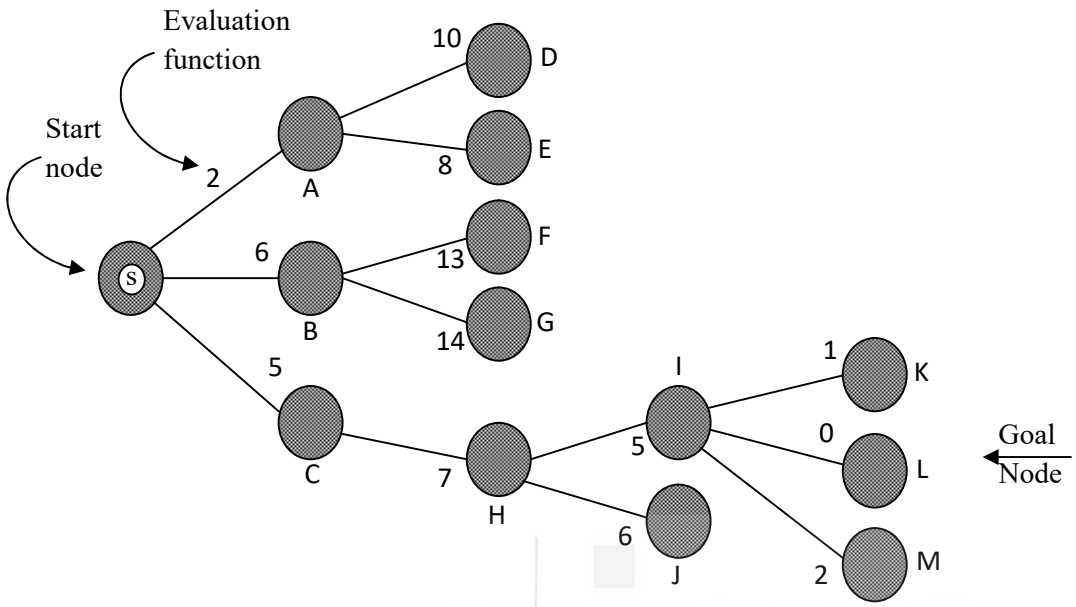


Fig. 2.17: Greedy best first search graph.

Working: We start with a start-nodes, S. Now, S has three children i.e., A, B and C with their Heuristic function values 2, 6 and 5 respectively. These weights show approximately, how far theyare from goal node. So, we write, children of S are -(A:2), (B: 6), (C:5)
Out of these, the node with minimum value is (A : 2). So, we select A and its children are explored (or generated).
Its children are(D: 10) and (E:8)
The search process now has four nodes to search for, namely-
(B : 6), (C: 5), (8.10) and (E: 8)
Out of these, node-C has the minimal value of 5. So, we select it and expand. So, we get(H: 7) as its child.
Now, the nodes to search are as follows-
(B:6), (D: 10,(E: 8) and (H : 7) and so on.

Working of the algorithm can be represented in tabular firm as follows

Step	**Node being expanded**	**Children (on expansion)**	**Available nodes (to search)**	**Node Chosen**
1.	S	(A:2), (B:6),(C:5)	(A:2), (B:6),(C:5)	(A:2)
2.	A	(D:10), (E:8)	(B:6),(C:5), (D:10), (E:8)	(C:5)
3.	C	(H:7)	(B:6), (D:10), (E:8),(H:7)	(B:6)
4.	B	(F:13), (G:14)	(D:10), (E:8), (H:7) ,(F:13), (G:14)	(H:7)
5.	H	(I:5), (J:6)	(D:10), (E:8),(F:13), (G:14), (I:5), (J:6)	(I:5)
6.	I	(K:1), (L:0), (M:2)	(D:10), (E:8), (H:7) ,(F:13), (G:14), (J:6),(K:1),(L:0),(M:2)	Goal node is found. So, search stops now

3.8 A* Algorithm

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 state g(n). It has combined features of uniform cost search (UCS) and greedy best-first search, by which it solves the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.
In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence, we can combine both costs as following, and this sum is called as a **fitness number (Evaluation Function)**.

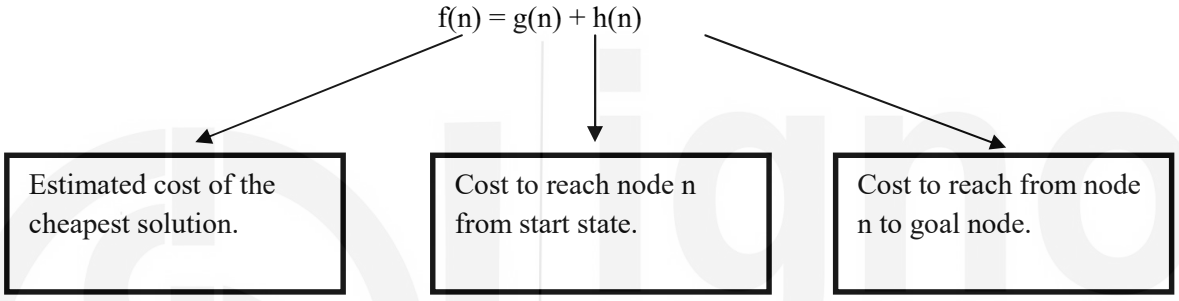


Fig 9 Evaluation function f(n) in A* Algorithm

A* algorithm **evaluation function** $\boldsymbol{f(n)}$is defined as $\boldsymbol{f(n) = g(n) + h(n)}$
Where $\boldsymbol{g(n)}$ =sum of edge costs from start state to n
And $\boldsymbol{h(n)}$ = estimate of lowest cost path from node n→goal node.

If h(n) is **admissible** then search will find optimal solution. Admissible means underestimates cost of any solution which can reached from node. In other words, a heuristic is called admissible if it always ***under-estimates***, that is, we always have $h(n) \leq h^*(n)$, where h*(n) denotes the minimum distance to a goal state from state n.
A* search begins at root node and then search continues by visiting the next node which has the least evaluation value f(n).
It evaluates nodes by using the following evaluation function
f(n) = h(n) +g(n) = estimated cost of the cheapest solution through n.
Where,
g(n): the actual shortest distance traveled from initial node to current node, it helps to avoid expanding paths that are already expansive
h(n): the estimated (or "heuristic") distance from current node to goal, it estimates which node is closest to the goal node.
Nodes are visited in this manner until a goal is reached.

Suppose s is a start state then calculation of evaluation function f(n) for any node n is shown in following figure 10.

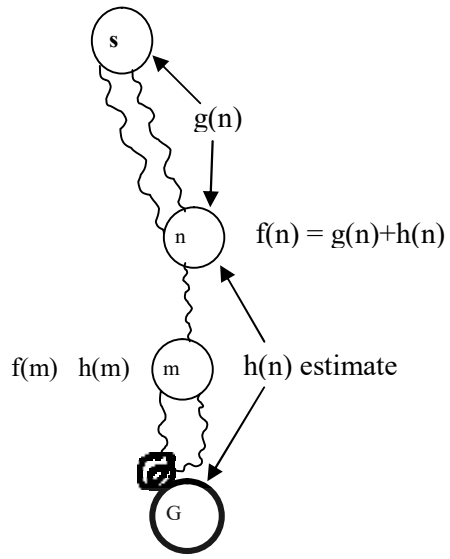


Fig 10 Calculation of evaluation function f(n).

Algorithm A*

1. **Initialize:** Set OPEN = (s), CLOSED = {}, g(s) = 0, f(s) = h(s)
2. **Fail:** If OPEN = { }, Terminate & fail
3. **Select** : Select the minimum cost state, n, from OPEN, Save n in CLOSED.
4. **Terminate**: If n ∈ G, terminate with success, and return f(n)
5. **Expand:** For each successor, m, of n

   If m ∉ [OPEN ∪ CLOSED]
   Set g(m) = g(n) + C(n,m)
   Set f(m) = g(m) +h(m)
   Insert m in OPEN

   If m ∈ [OPEN ∪ CLOSED]
   Set g(m) = min {min {g(m), g(n) + C(n,m)}
   Set f(m) = g(m) + h(m)

   If f(m) has decreased and m ∈ CLOSED,
   move m to OPEN

In step5, we generate a successor of n (say m) and for each successor m, if it does not belong to OPEN or CLOSED that is $m \notin [OPEN \cup CLOSED]$, then we insert it in OPEN with the cost g(n)+C(n,m) i.e., cost up to n and additional cost from n→m.
If $m \in [OPEN \cup CLOSED]$ then we set g(m) with original cost and new cost $[g(n) + C(n,m)]$. If we arrive at some state with another path which has less cost from original one, then we replace the existing cost with this minimum cost.
If we find f(m) is decreased (if larger then ignore) and $m \in CLOSED$ then move m from CLOSED to OPEN.

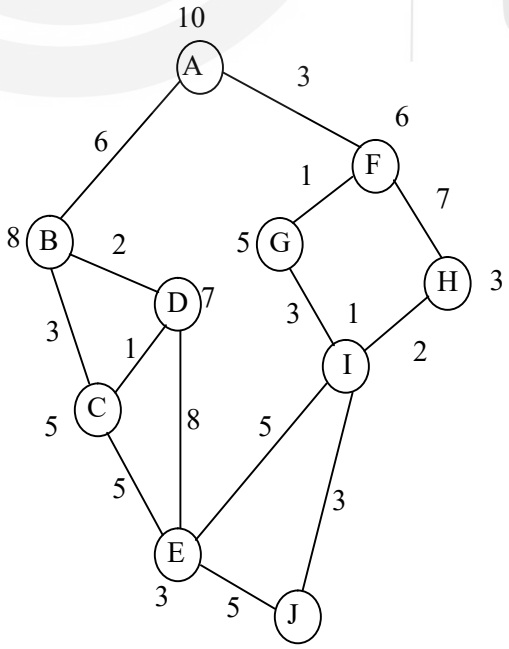
Note that, the implementation of A* Algorithm involves maintaining two lists- **OPEN** and **CLOSED**. The list **OPEN** contains those nodes that have been evaluated by the heuristic function but have not expanded into successors yet and the list **CLOSED** contains those nodes that have already been visited.

See the following steps for working of A* algorithm:

Step-1: Define a list OPEN. Initially, OPEN consists of a single node, the start node S.
Step-2: If the list is empty, return failure and exit.
Step-3: Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
If node n is a goal state, return success and exit.
Step-4: Expand node n.
Step-5: If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
Otherwise, go to Setp-6.
Step-6: For each successor node,
Apply the evaluation function f to the node.
If the node has not been in either list, add it to OPEN.
Step-7: Go back to Step-2.

3.8.1 Working of A* algorithm

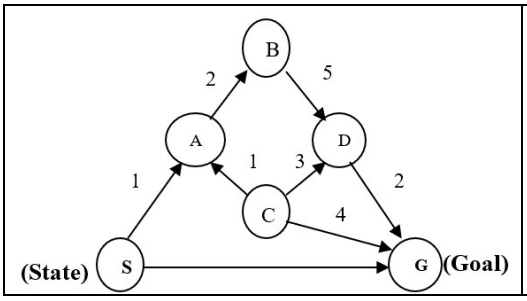
Example1: Let's us consider the following graph to understand the working of A* algorithm. The numbers written on edges represent the distance between the nodes. The numbers written on nodes represent the heuristic value. Find the most cost-effective path to reach from start state **A** to **final** state **J** using A* Algorithm.



Step-1:
We start with node A. Node B and Node F can be reached from node A. A* Algorithm calculates f(B) and f(F). Estimated Cost f(n) = g(n) +h(n) for Node B and Node F is:
f(B) = 6+8=14
f(F) = 3+6=9

Since f(F) < f(F), so it decides to go to node F.
➔ Closed list (F)
Path- A →F
Step-2: Node G and Node H can be reached from node F.
A* Algorithm calculates f(G) and f(H).
f(G) = (3+1)=5=9
f(H) = (3+7) +5=13
Since f(G) < f(H), so it decides to go to node G.
➔ Closed list (G)
Path- A→F→G
Step-3: Node I can be reached from node G.
A* Algorithm calculates f(I).
f(I)=(3+1+3)+1=8; It decides to go to node I.
➔ Closed list (I).
Path- A→F→G→I
Step-4: Node E, Node H and Node J can be reached from node I.
A* Algorithm calculates f(E), f(H), f(J).
f(E) = (3+1+3+5) + 3 =15
f(H) = (3+1+3+2) + 3 = 12
f(J) = (3+1+3+3) +0 = 10
Since f(J) is least, so it decides to go to node J.
➔ Closed list (J)
Shortest Path - A→F→G→I→J
Path Cost is 3+1+3+3=10

Example 2: Consider the following graph and apply A* algorithm and find the most cost-effective path to reach from start state **S** to **final** state**G**. The heuristic function value of each node n is defined in the table given.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:

S→A=1+3=4

S→G=10+0=10

S→A→B=1+2+4=7

S→A→C=1+1+2=4

S→A→C→D=1+1+3+6=11

S→A→C→G=1+1+4=6

S→A→B→D=1+2+5+6=14

S→A→C→D→G=1+1+3+2=7

S→A→B→D→G=1+2+5+2=10

3.8.2 Advantages and disadvantages of A* algorithm

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

3.8.3 Admissibility Properties of A* algorithm

A heuristic is called admissible if it always ***under-estimates***, that is, we always have $h(n) \leq h^*(n)$, where $h^*(n)$ denotes the minimum distance to a goal state from state n.For finite state spaces, A* always terminates.

In other words, **if the heuristic function h always underestimates then true cost h* (that is heuristic function cost h(n) is smaller than true cost h*(n)), then A* is guaranteed to find an optimal solution.**

If there is a path from s to a goal state, A* terminates (even when the state space is infinite)**.** Algorithm A* is admissible, that is, if there is a path from s to a goal state, A* terminates by finding an optimal path **.** If we are given two or more admissible heuristics, we can take there max to get a stronger admissible heuristic.

Admissibility Condition :

By admissible algorithm. we mean that the algorithm is sure to find a most optimal solution if one exists. Please note that this is possible only when the evaluation function value never overestimates the distance of the node to the goal. Also note that if the evaluation function value which is a heuristic one is exactly the same of the distance of the node to the goal, then this algorithm will immediately give the solution.

For example, the A* algorithm discussed above is admissible. There are three conditions to be satisfied for A to be admissible.
They are as follows-

1. Each node in the graph has finite number of successors (or O).
2. All arcs in the graph have costs greater than some positive amount, (say C).
3. For each node in the graph, n, h(n) ≤h'(n).

This implies that the heuristic guess of the cost of getting from node n to the goal is never an overestimate. This is known as a **heuristic condition.** Only if these three conditions are satisfied, A* is guaranteed to find an optimal (least) cost path. **Please note that A* algorithm is admissible for any node n if on such path, h'(n) is always less than or equal to h(n).** This is possible only when the evaluation function value never overestimates the distance of the node to the goal. Although the admissibility condition requires h'(n) to be a lower bound on h(n), it is expected that the more closely h'(n) approaches h(n), the better is the performance of the algorithm.

If h(n) = h'(n) -an optimal solution path would be found without over expanding a node of the path. We assume that one optimal solution exists. If h'(n)=0 then A* reduces to blind uniform cost algorithm or breadth-first algorithm.

Please note that the admissible heuristics are by nature optimistic because they think that the cost of solving the problem is less than it actually is because g(n) is the exact cost for each n. Also note that f(n) should never overestimate the true cost of a solution through n.
For example, consider a network of roads and cities with roads connecting these cities. Our problems to find a path between two cities such that the mileage/fuel cost is minimal. Then an admissible heuristic would be to use distance to estimate the costs from a given city to the goal city. Naturally, the air distance will be either equal to the real distance or will underestimate it i.e., h(n) ≤ h'(n).

3.8.4 Properties of heuristic Algorithm:

1. **Admissibility condition**: Algorithm A is admissible if it guarantees to return an optimal solution when one exists. A heuristic function h is called admissive if many general estimates., we always have $h(n) \leq h^*(n)$
2. **Completeness condition**: Algorithm A is complete if it always terminates with a solution when one exists.
3. **Dominance property**: If A_1 and A_2 are two Admissible versions of Algorithm A such that A is more informed than A_2 $h_1(n) > h_2(n)$.
4. **Optimal Property**: Algorithm A is optimal over a class of Algorithms If a dominates all members of the class.

3.8.5 Results on A* Algorithm

1. **A* is admissible: Algorithm A* is admissible** , that is, if there is a path from S to goal state, A* terminates by finding an optimal solution.
2. **A* is complete**: If there is a path from S to goal state, **A terminates** (Even when the state space is ∞).

3. **Dominance property**: If A_1& A_2→ two Admissible versions of A* S.t.
   A1 is more informed than A_2, then A_2 expends at least as many states as does A_1. (So A_1 dominates A_2 Here, b/s its better heuristics than A_2)

   If we are given two or more admissible heuristics, we can take their max to get a stronger admissible heuristic.

3.9 Problem Reduction Search

Problem reduction search is broadly defined as a planning how best to solve a problem that can be recursively decomposed into subproblems in multiple ways. There are many ways to decompose a problem, we have to find the best decomposition, which gives the quality of searching or cost is minimum.

We already know about the divide and conquer strategy, a solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its sub solution. These sub solutions can then be recombined to get a solution as a whole. That is called is **Problem Reduction**. This method generates arc which is called as AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved for an arc to point to a solution.

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. Consider the following example to understand the AND-OR graph (figure-11).

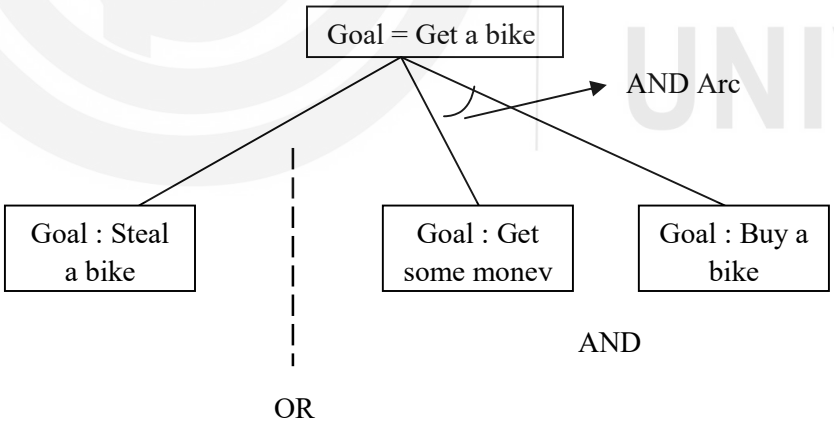


Fig 11**AND-OR graph**

The figure-11 shows an AND-OR graph. In an AND-OR graph, OR node represents a choice between possible decompositions, and an AND node represents given decomposition. For example, to Get a bike, we have two options, either:

1.(Steal a bike)

OR

2. Get some money **AND** Buy a Bike.

In this graph we are given two choices, first Steal a bike or get some money **AND** Buy a Bike. When we have more than one choice and we have to pick one, we apply OR condition to choose one.(That's what we did here).
Basically, the ARC here denotes AND condition.
Here we have replicated the arc between the Get some money and buy a bike because by getting some money possibility of buying a bike is more than stealing.
AO* search algorithm is based on AND-OR graph, so it is called AO* search algorithm. AO* Algorithm basically based on problem decomposition (Breakdown problem into small pieces).
The main difference between the A*(A star) and AO*(AO star) algorithms is that A* algorithm represents an OR graph algorithm that is used to find a single solution (either this or that). Butan **AO* algorithm** represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch.
A* algorithm guarantees to give **an optimal solution** while AO* doesn't since AO* doesn't explore all other solutions once it got a solution.

3.9.1 Problem definition in AND-OR graph:

Given **[G, s, T]**
Where **G:** Implicitly specified AND/OR graph
s: Strat node of the AND/OR graph
T: Set of terminal nodes (called SOLVED)
h(n): Heuristic function estimating the cost of solving the sub problem at n.

Example1:
Let us see one example with the presence of heuristic value at every node (see fig 2) . The estimated heuristic value is given at each node. The heuristic value h(n) at any node indicates "from this node at least h(n) value (or cost) is required to find solution". Here we assume the edge cost value (i.e., g(n) value) for each edge is 1. Remember in OR node we always mark that successor node which indicates best path for solution.

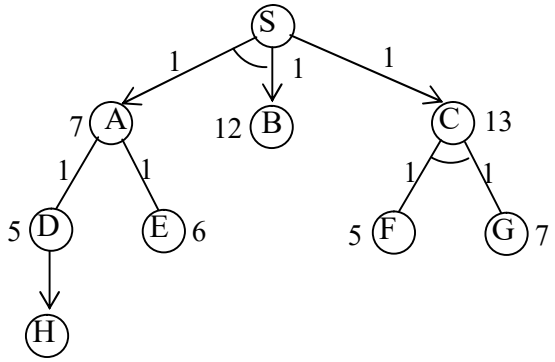
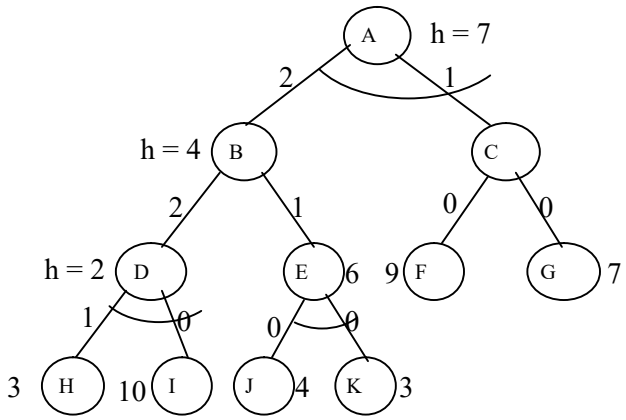


Fig 12: AND-OR graph with heuristic value at each node

Note that, the graph given in fig 2, there are two paths for solution from start state S: either S-A-B or S-C. To calculate the cost of the path we use the formula f(n)=g(n)+h(n) [note that here g(n) value is 1 for every edge].
Path1: f(S-A-B)=1+1+7+12=21
Path2: f(S-C)=1+13=14

Since $min(21,14) = 14$; so, we select successor node C, as its cost is minimum, so it indicates best path for solution.
Note that C is a AND node; so, we consider both the successor node of C. The cost of node C is f(C-F-G)=1+1+5+7=14; so, the revised cost of node C is 14 and now the revised cost of node S is f(S-C)=1+14=15 (revised).
Note that once the cost (that is f value) of any node is revised, we propagate this change backward through the graph to decide the current best path.
Now let us explore another path and check whether we are getting lessor cost as compared to this cost or not.
F(A-D)=1+5=6 and f(A-E)=1+6=7; since A is an OR node so best successor node is D since min(6,7)=6. So revised cost of Node A will be 6 instead of 7, that is f(A)=6 (revised). Now next selected node is D and D is having only one node H so f(D-H)=1+2=3, so the revised cost of node D is 3, so now the revised cost of node A, that is f(A-D-H)=4. This path is better than f(A-E)=7. So, the final revised cost of node A is 4. Now the final revised cost of f(S-A-B)=1+1+4+12=18 (revised).
Thus, the final revised cost for
Path1: f(S-A-B)=18 and
Path2: f(S-C)=15
So optimal cost is 15.

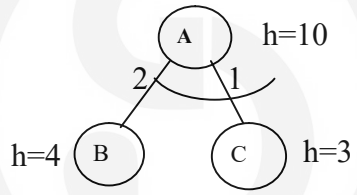
Example2:
Consider the following AND-OR Graph with estimated heuristic cost at every node. Note that A,D,E are AND node and B, C are OR node. Edge cost (i.e., g(n) value) is also given.
Apply AO* algorithm and find the optimal cost path using AO* algorithm.



A,D,E → AND Node
B,C → OR Node

Fig- 1: AND-OR graph with heuristic function value.

Heuristic (Estimated) cost at every Node is given. For example, heuristic cost at node A is h=7, which means at least 7-unit cost required to find a solution.
Since A is AND Node, so we have to solve Both of its successor Node B and C.
Cost of Node A i.e., f(A-B-C) = (2+4) + (1+3) = 10
We perform cost revision in Bottom-up fashion.



i.e., heuristic value of Node A will be 10.

Fig (a)Cost revision

Now, Let's see R.H.S first, Node C is an OR Node.
So, cost f(C-F)= 0+9=9
& f(C-G) =0+7=7
So Best successor of C in G, so we perform cost revision in Bottom-up fashion as follows:

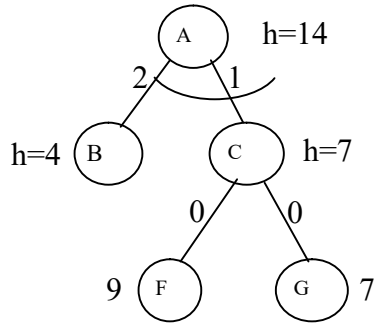


Fig (b): Cost revision in Bottom-up.

Now revision cost of Node A is 14; So, till now, we can say that the best path to solve the problem is: G-C-A. Now expend left node of root, i.e., Node B.

Since B is an OR Node; So, we have to See, which successor is best; i.e., D or E.
The revised cost of Node D is = (3+1) +(10+0) =14
And the revised cost of Node E = (40+0)+(3+0) = 7
So, the Best promising successor Node is E.
Now, perform cost Revision in Bottom-up fashion.

Note that all the leaf Node in the Marked tree is solved is solved. So, the best way to solve the problem is to following the marked tree and solving those marked problem. This best cost to solve the problem is **18.** Note that for **AND Node**: if both successor (unit problem) is solved, then we declared SOLVED and for **OR Node**: if any one best successor is SOLVED, then we declared SOLVED.

3.9.2 AO* algorithm

Our real-life situations cannot be exactly decomposed into either AND tree or OR tree but is always combination of both. So, we need an AO* algorithm where O stands for 'ordered'. Instead of two lists OPEN and CLOSED of A* algorithm, we use a single structure GRAPH in AO* algorithm. If represents a part of the search graph that has been explicitly generated so far. Please note that each node in the graph will point both down to its immediate successors and to its immediate predecessors. Also note that each node will have some h'(n) value associated with it. But unlikeA* search, g(n) is not stored. It is not possible to compute a single value of g(n) due to many paths to the same state. It is not required also as we are doing top-down traversing along best-knownpath.
This guarantees that only those nodes that are on the best path are considered for expansion
Hence, h'(n) will only serve as the estimate of goodness of a node.
Next, we develop an AO* algorithm.

Algorithm AO*

1. Initialize: Set G* ={s}, f(s) =h(s)
   If s ∈ T, label s as SOLVED
2. Terminate: If s is SOLVED, then Terminate
3. Select: Select a non-terminal leaf node n from the marked
   sub-free
4. Expand: Make explicit the successors of n for each new successors, m:
   Set f(m) =h(m)
   If m is terminal, label m SOLVED
5. Cost Revision: Call C**ost-Revise (n)**
6. Loop: Go to Step 2.

Cost Revision in AO*: Cost-Revise(n)

1. Create Z = {n}
2. If Z = {} return
3. Select a node m from Z such that m has no descendants in Z
4. If m is an AND node with successors $r1, r2, \ldots rk$:

Set $f(m) = \sum [f(r_i) + c(m, r_i)]$
Mark the edge to each successor of m
If each successor is labelled SOLVED,
then label m as SOLVED.

5. If m is an OR node with successor $r1, r2, \ldots rk$:

Set $f(m) = min\{f(r_i) + c(m, r_i)\}$
Mark the edge to the best successor of m
If the marked successor is labelled SOLVED, label m as SOLVED

6. If the cost or label of m has changes, then insert those parents of m into Z for which ma is a marked successor

3.9.3 Advantage of AO* algorithm

Note that **AO* will always find a minimum cost solution if one exists if** $h'(n) < h(n)$ and that all arc costs are positive. The efficiency of this algorithm will depend on how closely $h'(n)$ approximates $h(n)$. Also note that **AO* is guaranteed to terminate** even on graphs that have cycles**.**

Note: When the graph has only OR node then AO* algorithm works just like A* algorithm.

3.10 Memory Bound Heuristic Search

The following are the commonly used memory bound heuristics search:

1. Iterative deepening A* (IDA*)
2. Recursive Best-First search (RBFS)
3. Memory bound A* (MBA*)

3.10.1: Iterative Deepening A* (IDA*)

IDA* is a variant of the A* search algorithm which uses iterative deepening to keep the memory usage lower than in A*.It is an informed search based on the idea of the uniformed iterative deepening search. Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:

The depth bound modified to be an f-limit
1. Start with limit = h(start)
2. Prune any node if f(node) > f-limit
3. Next f-limit = minimum cost of any node pruned

Iterative Deepening is a kind of uniformed search strategy. It combines the benefits of depth-first and breadth- first search.

Advantage of IDA* is: It is optimal and complete like breadth first search and modest memory requirement like depth-first search.

IDA* algorithm

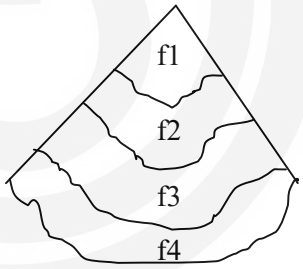
1. Set C= f(s)
2. Perform DFBB with cut-off C
   Expand a state, n, only if its f-value is less than or equal to C
   If a goal is selected for expansion, then return C and terminate
3. Update C to the minimum f-value which exceeded C among states which were examined and Go TO Step 2.

3.10.2 Working of IDA*

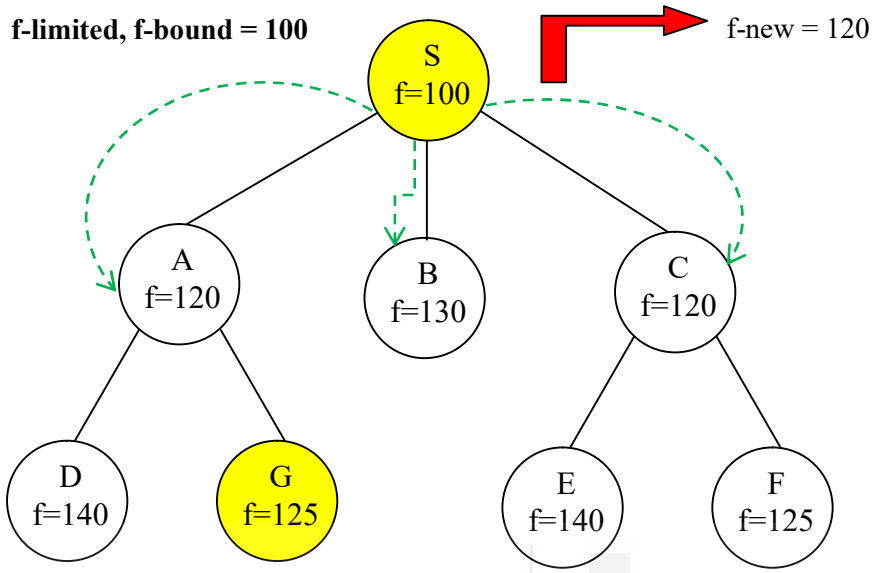
- Perform depth-first search LIMITED to some f-bound.
- If goal found then ok.
- Else: increase the f-bound and restart.

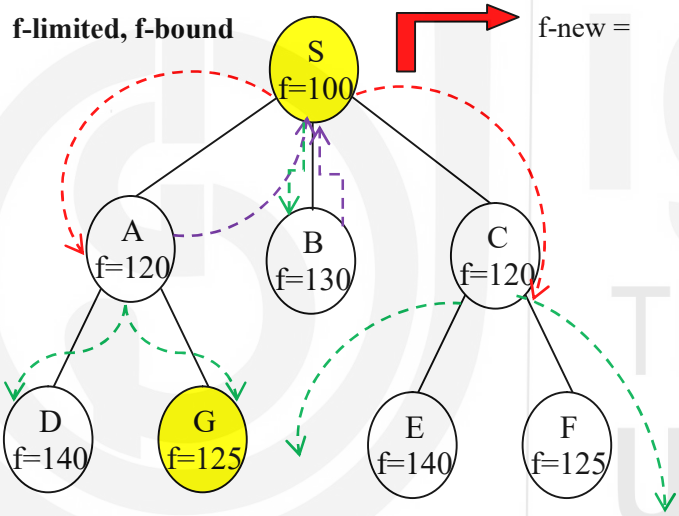
How to establish the f-bound?
Initially: f(S)

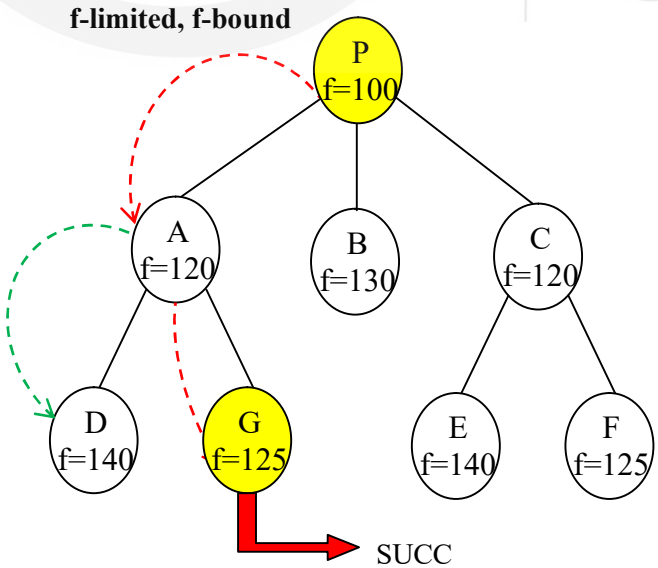
- Generate all successors
- Record the minimal f(succ) > f(S)
- Continue with minimal f(succ) instead of f(S)



Consider the following example to understand the IDA*


f-limited, f-bound = 100
S
f=100
f-new = 120
A
f=120
B
f=130
C
f=120
D
f=140
G
f=125
E
f=140
F
f=125


f-limited, f-bound
S
f=100
f-new =
A
f=120
B
f=130
C
f=120
D
f=140
G
f=125
E
f=140
F
f=125


f-limited, f-bound
P
f=100
A
f=120
B
f=130
C
f=120
D
f=140
G
f=125
E
f=140
F
f=125
SUCC

3.10.3 Analysis of IDA*

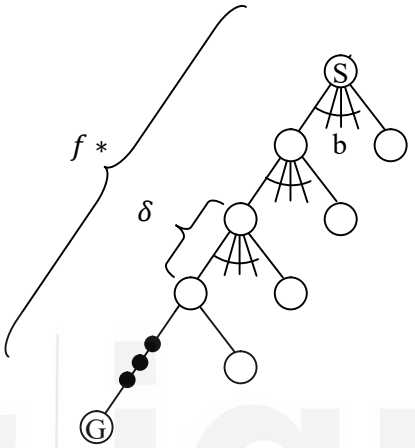
IDA* is complete, optimal, and optimally efficient (assuming a consistent, admissible heuristic), and requires only a polynomial amount of storage in the worst case:

F* = optimal path cost to a goal

B = branching factor

δ= minimum operator step cost

$\frac{bf*}{\delta}$nodes of storage required



Note that IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.

3.10.4 Comparison of A* and IDA* algorithm:

- Iterative Deepening Search (IDS) is nothing but BFS plus DFS for tree search.
- IDA* algorithm is "complete and Optimal" algorithm.
- BFA and A* is good for optimality, but not memory.
- DFS: good for memory O(bd), but not optimality
- In the worst case, only one new state is expanded in each iteration
- (IDA*). If A* expands N states, then IDA* can expand:

  $1+2+3+\ldots+N=O(N^2)$

3.11 Recursive Best first search (RBFS)

The idea of recursive best first search is to simulate **A*** search with **O(bd)** memory, where b is the branching factor and d is the solution depth.

It is a memory bound, simple recursive algorithm that works like a standard best first search but only takes up linear space. There are some things that make it different from recursive DFS. It keeps track of f, the value of the best alternative path that can be found from any ancestor of the current node, instead of continuing indefinitely down the current path.

RBFS mimic the operation of standard Best-First search algorithm. IBFS keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds the limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its

children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time.
RBFS is somewhat most efficient than IDA*, but still suffers from excessive node regeneration.
A* and RBFS are optimal algorithms if heuristic function h(n) is admissible.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

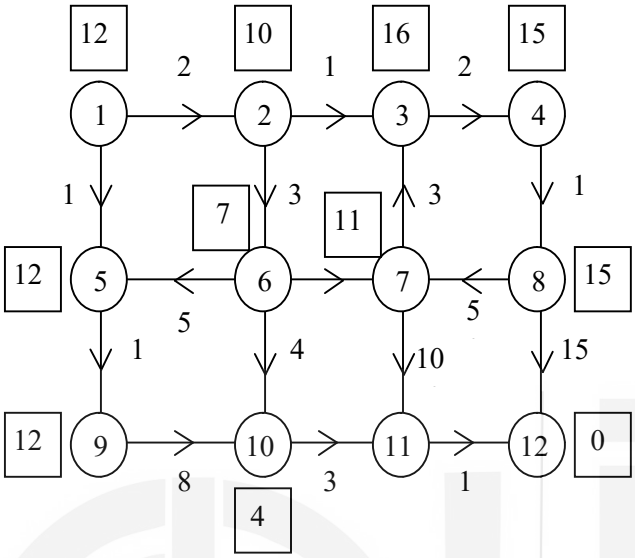
Fig12 Algorithm for recursive Best first Search

3.11.1 Advantages and Disadvantages of RBFS:

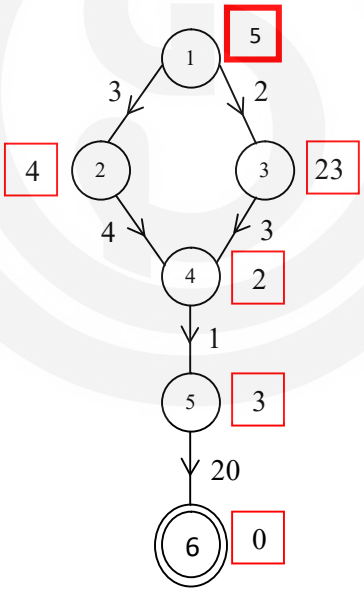
- More efficient than IDA* and still optimal.
- Best-first Search based on next best f-contour; fewer regeneration of nodes.
- Exploit results of search at a specific f-contour by saving next f-contour associated with a node who successors have been explored.
- Like IDA* still suffers from excessive node regeneration.
- IDA* and RBFS not good for graphs.
- Can't check for repeated states other than those on current path.
- Both are hard to characterize in terms of expected time complexity.

☞ Check Your Progress 2

Q.1 Apply A* on the following graph:-

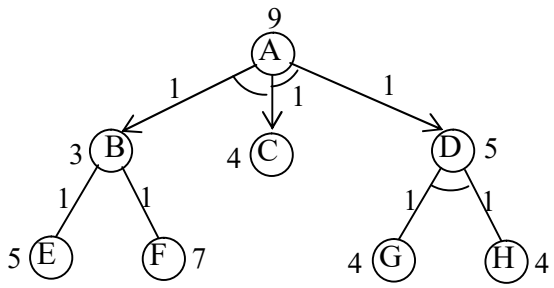


Q.2 Apply A* algorithm on the following graph

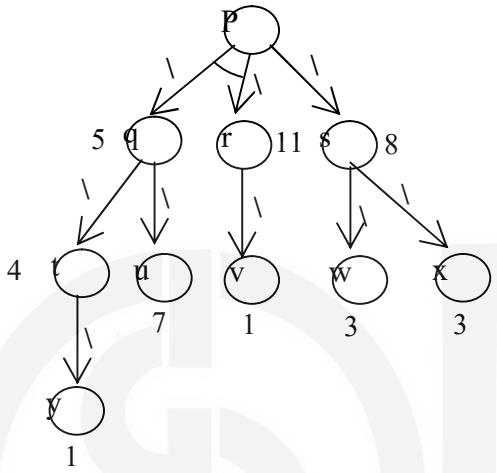


Q.3 Differentiate between the A* and AO* algorithm.

Q.4: Apply AO* algorithm on the following graph. Heuristic value is also given at every node and assume the edge cost value of each node is 1.



Q.5 Apply AO* algorithm on the following graph. Heuristic value is also given at every node and assume the edge cost value of each node is 1.



Example 6: Given the 3 matrices A1, A2, A3 with their dimensions $(3 \times 4), (4 \times 10), (10 \times 1)$. Consider the problem of solving this **chain matrix multiplication**. Apply the concept of AND-OR graph and find a minimum cost solution tree.

(Multiple choice Questions)

Q.6 A* algorithm always to find an optimal solution if

A. h' is always 0
B. g is always 1
C. h' never overestimates h
D. h' never underestimate h

Q.7A* algorithm uses $f^* = g + h^*$ to estimate the cost of getting from the initial state to the goal state, where g is a measure of cost getting from initial state to the current node and the function h^* is an estimate of the cost of getting from the current node to the goal state. To find a path involving the fewest number of steps, we should test,

A. g=1
B. g=0
C. $h^* = 0$
D. $h^* = 1$

3.12 Summary

- As the name 'Uninformed Search' means the machine blindly follows the algorithm regardless of whether right or wrong, efficient or in-efficient.
- These algorithms are brute force operations, and they don't have extra information about the search space; the only information they have is on how to traverse or visit the nodes in the tree. Thus, uninformed search algorithms are also called **blind search** algorithms.
- The search algorithm produces the search tree without using any domain knowledge, which is a brute force in nature. They don't have any background information on how to approach the goal or whatsoever. But these are the basics of search algorithms in AI.
- **The different types of uninformed search algorithms are as follows:**
    - Depth First Search
    - Breadth-First Search
    - Depth Limited Search
    - Uniform Cost Search
    - Iterative Deepening Depth First Search
    - Bidirectional Search (if applicable)
- The following terms are frequently used in any search algorithms:

- **State:** It provides all the information about the environment.
- **Goal State**: The desired resulting condition in a given problem and the kind of search algorithm we are looking for.
- **Goal Test:** The test to determine whether a particular state is a goal state.
- **Path/Step Cost:** These are integers that represent the cost to move from one node to another node.

- To evaluate and compare the efficiency of any search algorithm, the following 4 properties are used:

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution, if exist.
- **Optimality/Admissibility:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Space Complexity:** A function describing the amount of space(memory) an algorithm takes in terms of input to the algorithm. That is how much space is used by the algorithm? Usually measured in terms of the maximum number of nodes in memory at a time.
- **Time Complexity:** A function describing the amount of time the algorithm takes in terms of input to the algorithm. That is, how long (worst or average case) does it take to find a solution?
- Time and space complexity are measured in terms of:'b' – maximum branching factor (Max number of successor (child) of any node) in a tree, 'd' – the depth of the shallowest goal node, and 'm' – maximum depth of the search tree (maybe infinity).
- The following table summarizes the 4 properties(**Completeness, Optimality/Admissibility,SpaceComplexity, Time Complexity)** of the search algorithm.

	BFS	DFS	IDDFS	Bidirectional Search (if Applicable)
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

- The advantage of DFS is it requires very little memory as compared to BFS, as it only needs to store a stack of the nodes on the path from the root node to the current node. The disadvantages of DFS are as follows: There is the possibility that many states keep reoccurring, and there is no guarantee of finding the solution. The DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.
- On the other hand, BFS gives a guarantee to get an optimal solution, if any solution exists (Completeness) and if there is more than one solution for a given problem, then BFS will provide the minimal solution which requires the least number of steps (Optimal), but one major drawback of BFS is that it requires lots of memory space since each level of the tree must be saved into memory to expand the next level.
- Iterative Deepening DFS (IDDFS) combines the benefits of both BFS and DFS search algorithms in terms of fast search and memory efficiency. It is better than DFS and needs less space than BFS. But the main drawback of IDDFS is that it repeats all the work from the previous phase.
- The advantage of Bidirectional search (BS) is that it uses various techniques like DFS, BFS, DLS, etc, so it is efficient and requires less memory. The Implementation of the bidirectional search tree is difficult and in bidirectional search, one should know the goal state in advance.
- In Informed search the domain dependent (heuristic) information is used in order to search the space more efficiently.
- Informed searched include the following search:
- **Best-first search:** Order agenda based on some measure of how 'good' each state is.
- **Uniform-cost search**: Cost of getting to current state from initial state = g(n)
- **Greedy search:** Estimated cost of reaching goal from current state – Heuristic evaluation functions, h(n)
- **A* search:** f(n) = g(n) + h(n)
- Admissibility: h(n)never overestimates the actual cost of getting to the goal state.
- Informed Ness: A search strategy which searches less of the statesperson order to find a goal state is more informed.
- A* algorithm avoid expanding paths that are already expensive.
- In A*, Evaluation function f(n) = g(n) + h(n), where g(n) = cost so far to reach n, h(n) = estimated cost to goal from n, f(n) = estimated total cost of path through n to goal.
- A∗ search uses an admissible heuristic function, i.e., $h(n) \leq h*(n)$ where $h*(n)$ is the true cost of cheapest solution from n.
- A∗ search has a very good property: A∗ search is optimal! So, if there is any solution, A∗ search is guaranteed to find a least cost solution. Remember, this needs an admissible heuristic.

- The commonly used memory bound heuristics search are Iterative deepening A* (IDA*), Recursive Best-First search (RBFS) and Memory bound A* (MBA*).
- Iterative Deepening Search (IDS) is nothing but BFS plus DFS for tree search. IDA* algorithm is "complete and Optimal" algorithm.
- The idea of recursive best first search is to simulate **A*** search with **O(bd)** memory, where b is the branching factor and d is the solution depth.

3.13 Solutions/Answers

Check your progress 1:

Answer 1:

Table 1: Open and closed list for BFS

OPEN LIST	CLOSED LIST
1	1
2,3,4	2
3,4,6	3
4,6,5	4
6,5,7	5
5,7	6
7,	7
8-Goal State	-

Table 2: Open and closed list for DFS

OPEN LIST	CLOSED LIST
1	1
4,3,2	2
4,3,6	6
4,3	3
4,5	5
4,7	7
4,8-Goal State	-

Answer 2

1. **open = [A]; closed = [ ]** B is not the goal.
2. **open = [B,C,D]; closed = [A]** Put his children onto the queue.
3. **open = [C,D,E,F}; closed = [B,A]** Put him in closed. He is done.
4. **open = [D,E,F,G,H]; closed =[C,B,A]** Items between red bard are siblings.
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**

6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M,N]** (as L is already on open);**closed = [G,F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. and so on until either goal is reached or open is empty.

Answer 3: Refer BFS section for solution.

Answer 4

	BFS	**DFS**	**IDDFS**	**Bidirectional Search (if Applicable)**
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

Check your progress 2

Answer 1:
The OPEN and CLOSED list are shown below. Node with their f(n) values are inserted in OPEN list and that node will be expended next whose f(n) value is minimum.

CLOSED

$1^{(12)}$	$2^{(12)}$	$6^{(12)}$	$5^{(13)}$	$10^{(13)}$	$11^{(13)}$	$12^{(13)}$

OPEN

$1^{(12)}$			
$2^{(12)}$	$5^{(13)}$		
$5^{(13)}$	$3^{(14)}$	$6^{(12)}$	
$5^{(13)}$	$3^{(14)}$	$7^{(17)}$	$10^{(13)}$
$3^{(19)}$	$7^{(17)}$	$10^{(13)}$	
$3^{(19)}$	$7^{(17)}$	$10^{(13)}$	$9^{(14)}$
$3^{(19)}$	$7^{(17)}$	$9^{(14)}$	$11^{(13)}$
$3^{(19)}$	$7^{(17)}$	$9^{(14)}$	$12^{(13)}$

Note that only 6 nodes are expended to reach to a goal node. Optimal cost to reach from start state (1) to goal node (12) is **13**.

Note: If all the edge cost is positive then **Uniform cost search** (UCS) algorithm is same as Dijkstra's algorithm. Dijkstra algorithm fails if graph is having a negative weight cycle. A* algorithm allows negative weight also. It means A* algorithm work for negative (-ve) edge cost also. If some edge cost is

negative, then at any point of successive iteration, we cannot say till that node we have optimum cost (because of the negative cost). So, in this case (-ve edge cost), nodes come back from CLOSED to OPEN. Let us see one example (Example 2) having negative edge cost and you can also see how nodes come back from CLOSED to OPEN.

Answer 2:
The OPEN and CLOSED list are shown below. Node with their f(n) values are inserted in OPEN list and that node will be expended next whose f(n) value is minimum.

CLOSED

$1^{(15)}$ $2^{(7)}$ ~~$4^{(9)}$~~ ~~$5^{(11)}$~~ $3^{(25)}$ $4^{(7)}$ $5^{(9)}$

OPEN

$1^{(15)}$	
$2^{(7)}$	$3^{(25)}$
$3^{(25)}$	$4^{(9)}$
$3^{(25)}$	$5^{(11)}$
$3^{(25)}$	$6^{(28)}$
$4^{(7)}$	$6^{(28)}$
$5^{(9)}$	$6^{(28)}$
$6^{(26)}$	$6^{(28)}$

This is goal Node, but we can't Pick this because better cost/path may exist.
(Note: In open $6^{(28)}$ is Not min. cost ∴ we do not pick them).

Cost of 4 is decreased from 9 →7 so bring 4 from close →open
∴$4^{(9)}$

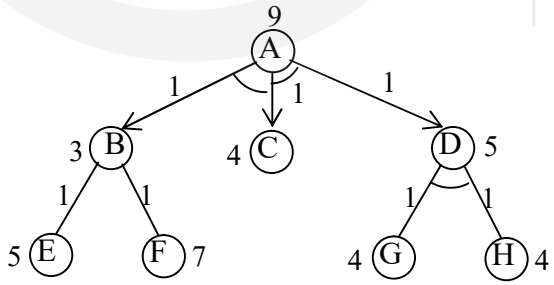
Same Logic

Goal (optimal cost is 26)

Optimal cost to reach from start state (1) to goal node (6) is **26**.

Answer 3: An A* algorithm represents an OR graph algorithm that is used to find a single solution (either this or that), but an AO* algorithm represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch.

Answer 4 :

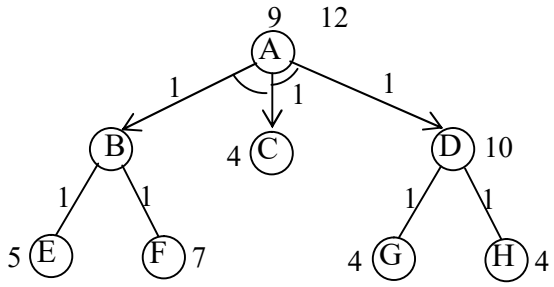


Path – 1: f(A-B-C) = 1+1+3+4=9
f(B-E) = 1+5=6 f(B-F) = 1+7=8
f(A-B-C) = 1+1+6+4=12

Path-2 : f(A-C-D) = 1+1+4+5=11
f(D-G-H) = 1+1+4+4=10

f(A-C-D)= 1+1+4+10=16

AND-OR graph with revised cost is shown in figure.



So, the optimal cost f(A-B-C)= 12

Note that AO* algorithm does not explore all the solution path once it finds a solution.

Answer 5: Similar to Q.4

Answer 6:

Given a chain of matrices: $A1, A2, \ldots\ldots, An$, where each matrix A_i has a dimension $p_{i-1} \times p_i$. Problem is to determine the order of multiplication or the way of parenthesizing the product of matrices, so that the minimum number of required operations (i.e., multiplications) can be minimized.

There are only 2 ways to parenthesizing the given 3 matrices, A1, A2, A3:

$A1 \times (A2 \times A3)$

or

$(A1 \times A2) \times A3$

As we know, if matrix A is of dimension $(p \times q)$ and matrix B is of dimension $(q \times r)$, then the cost of multiplying A to B that is $(A \times B)$ is $(p \times q \times r)$ and the final dimension of the resultant matrix $(A \times B)$ is $(p \times r)$.

Let us see how AND-OR graph is used to get the solution of this problem.

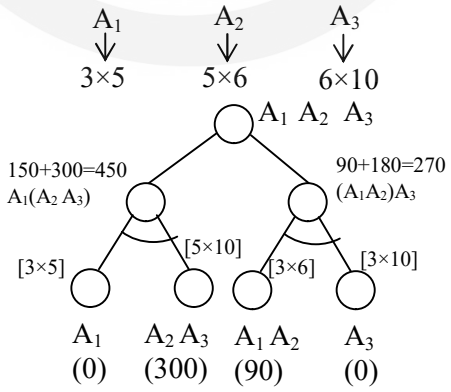


Figure-1 AND/OR graph for multiplying $A1 \times A2 \times A3$

In this AND-OR graph, parent (root) node indicates the given problem for multiplying A1A2A3. Next level of the tree (2 successors node) indicates the 2 choices (or ways) of multiplying (or

parenthesizing) the A1A2A3; first way is $A1 \times (A2 \times A3)$ and another way is $(A1 \times A2) \times A3$. Since out of these two choices, anyone will be the solution so there is a OR node for this. In an OR node, we always mark the current best successor node. Next level we have an AND node. For any AND node we must add the cost of both the successor node.
Cost of multiplying $(A2 \times A3) = 5 \times 6 \times 10 = 300$ and dimension of $A2 \times A3$ is (5×10). Since the dimension of A1 is (3×5) and the dimension of $A2 \times A3$ is (5×10), so the cost of multiplying $A1 \times (A2 \times A3) = 3 \times 5 \times 10 = 150$. Thus the total cost will be 300+150=450. Similarly,
The cost of multiplying $(A1 \times A2) = 3 \times 5 \times 6 = 90$ and dimension of $A1 \times A2$ will be (3×6). Since the dimension of $A1 \times A2$ is (3×6) and the dimension of A3 is (6×10), so the cost of multiplying $(A1 \times A2) \times A3 = 3 \times 6 \times 10 = 180$. Thus the total cost will be 90+180=270. So, the best way to multiplying $A1 \times A2 \times A3$ is $(A1 \times A2) \times A3$ and the minimum cost of multiplying $A1 \times A2 \times A3$
is 270.

Multiple Choice Questions

Answer 6: Option C

Answer 7: Option A

3.14 FURTHER READINGS

1. Ela Kumar, " Artificial Intelligence", IK International Publications
2. E. Rich and K. Knight, "Artificial intelligence", Tata Mc Graw Hill Publications
3. N.J. Nilsson, "Principles of AI", Narosa Publ. House Publications
4. John J. Craig, "Introduction to Robotics", Addison Wesley publication
5. D.W. Patterson, "Introduction to AI and Expert Systems" Pearson publication

s