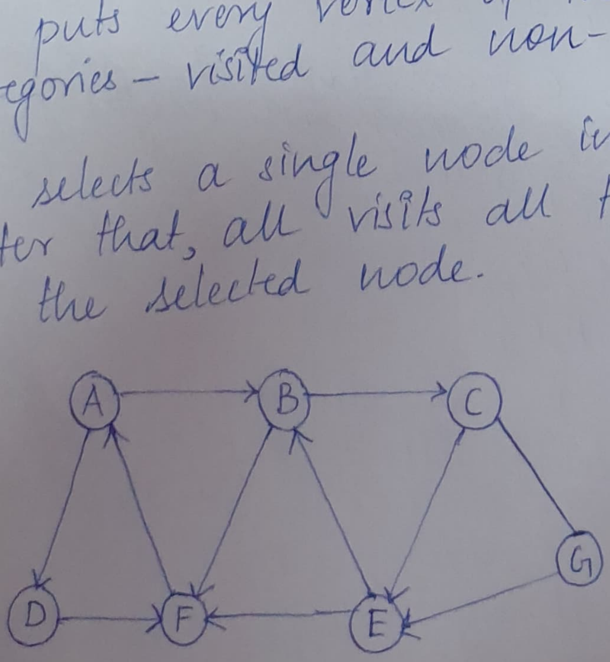
Rakesh Raja K B, B.Sc Cs D Sec
RA1931005020188

1. Explain in brief along with example Following types of uninformed search algorithms:
   - Breadth-first search
   - Depth-first Search
   - Depth-limited Search
   - Iterative deepening depth-first search
   - Uniform cost search
   - Bidirectional Search

Breadth-first algorithm

- Breadth first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes.
- Then, it selects the nearest node and explores all the unexplored nodes.
- While using BFS for traversal, any node in the graph can be considered as the root node.
- BFS is the most commonly used approach. It is a recursive algorithm to search all vertices of a tree or graph data structure.
- BFS puts every vertex of the vertex into two categories - visited and non-visited
- It selects a single node in a graph and after that, all visits all the nodes adjacent to the selected node.

eg:



Step 1: First, add A to queue 1 and NULL its queue 2

Queue 1 = {A}
Queue 2 = {NULL}

Step 2: Now, delete node A from queue 1 and add into queue 2. Insert all neighbors of node A to queue 1.

Queue 1 = {B, D}
Queue 2 = {A}

Step 3: Now, delete node B from queue 1 and add into queue 2. Insert all neighbors of node B to queue 1.

Queue 1 = {D, C, F}
Queue 2 = {A, B}

Step 4: Now, delete node D from queue 1 and add into queue 2. Insert all neighbors of node D to queue 1. The only neighbor of node D is F since it is already inserted, so it will not be inserted again.

Queue 1 = {C, F}
Queue 2 = {A, B, D}

Step 5: Delete node C from queue 1 and add into queue 2. Insert all neighbors of node C to queue 1.

Queue 1 = {F, E, G}
Queue 2 = {A, B, D, C}

Step 6: Delete node F from queue 1 and add into queue 2. Insert all neighbors of node F to queue 1. Since all the neighbors of node F are already present, we will not insert again.

Queue 1 = {E, G}
Queue 2 = {A, B, C, D, F}

Step 7: Delete node E from queue 1. Since all of its neighbors have already been added, so we will not insert again. Now, all the nodes are visited, and the target node E is encountered into queue 2.

Queue 1 = {G}
Queue 2 = {A, B, C, D, F, E}

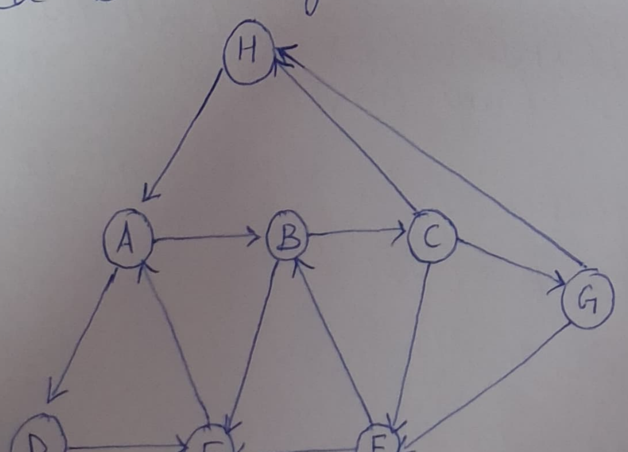
• Depth First search

• Depth-first search algorithm starts with initial node of the graph G, then goes to deeper until we find the goal node or node which has no children.

• The algorithm, then backtracks from the dead end towards the most recent node that is yet to be compelety unexplored.

• The data structure which is being used in DFS in stack.

• In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.



Push H onto the stack

Stack : H

POP to the top element of the stack i.e, H, print it and push all the neighbors of H onto the stack that is ready state.

Print H

Stack : A

POP the top element of the stack, i.e A, print it and push all the neighbors of A onto the stack that are in ready state.

Print A

Stack : B, D

POP the top element of the stack, i.e F, print it and push all the neighbors of F onto the stack that are in ready state

Print F

Stack ; B

POP the top of the stack i.e B and push all the neighbors

Print B

Stack : C

POP the top of the stack i.e C and push all the neighbors

Print C

Stack : E, G

POP the top of the stack i.e G and push all the neighbors

Print E

Stack

Hence, the stack becomes empty and all the nodes of the graph have been ~~tran~~ traversed.

The printing sequence of the graph, will be

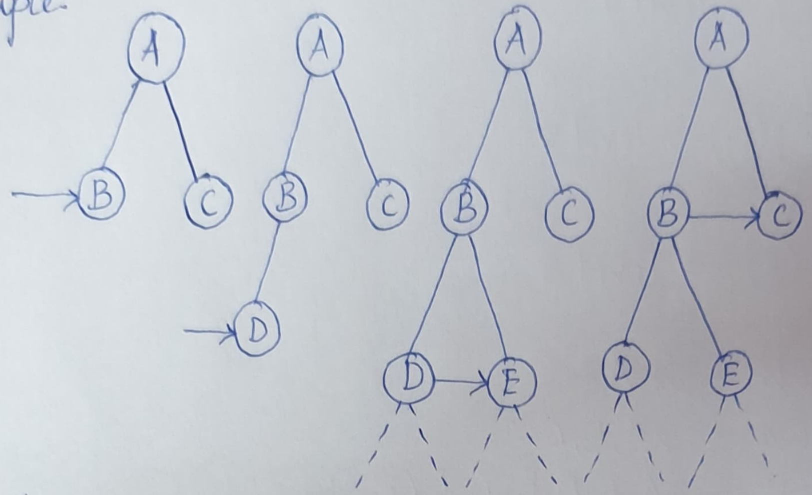
H → A → D → F → B → C → G → E

- <u>Depth-Limited Search Algorithms</u>

- A depth limited search algorithm is similar to depth-first search with a predetermined limit

- Depth-limited search can solve the drawback of the infinited path in the Depth-first search algorithm.
- In this algorithm, the node at the depth limit will treat as it has ~~to~~ no successor nodes further

Example:

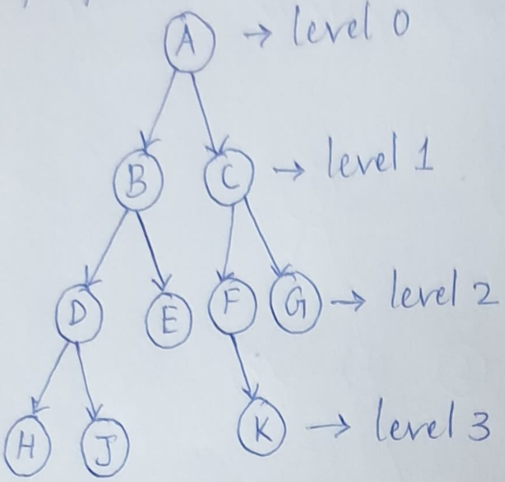


Algorithm

1) We start with finding and fixing a start node
2) Then we search along with the depth using the DFS Algorithm
3) Then we keep checking if the current node is the goal node or not.

- <u>Iterative deepening depth-first search</u>
- The Iterative deepening depth-first search as combination of DFS and BFS Algorithm. This search finds out the best depth limit and ~~doesn't~~ does it by gradually increasing the limit goal is found.
- This Algorithm performs depth first ~~and~~ search up to certain "depth limit" and it keeps increasing the depth limit after each iteration until the goal node is found.
- This search Algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

eg:
IDDFS algorithm performs various iterations until it doesn't find the goal node. The iteration performed by the algorithm is given as

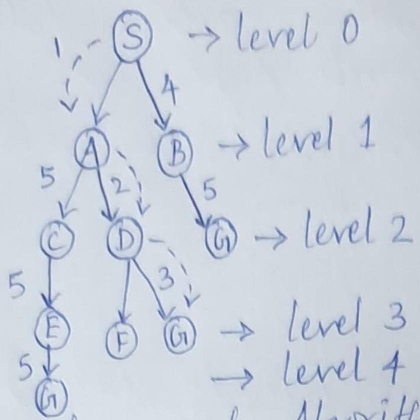


1st iteration --→ A
2nd iteration --→ A, B, C
3rd iteration → A, B, D, E, C, F, G
4th iteration → A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node

- <u>Uniform cost search Algorithm</u>
- uniform cost search is searching used for a traversing a weighted tree or graph. This Algorithm comes into play when a different cost is available for each edge.
- The primary goal of the uniform-cost search is ~~ta~~ to find the goal node which has the lowest cumulative cost
- uniform-search expands nodes according to their path costs from the root node.
- It can be used to solve any graph/tree which the optimal cost in demand.

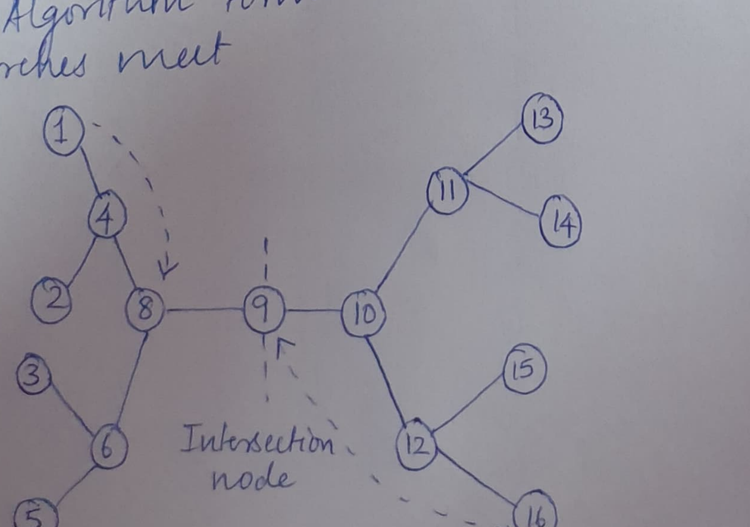
eg:



- **Bidirectional search Algorithm**
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other form goal node called as backward-search, to find the goal node
- Bidirectional search replaces one single search graph with two subgraphs in which one starts the search form an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other

Eg: In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 to forward direction and starts from goal node 16 in the backward direction.

- The Algorithm terminates at node 9 where two searches meet



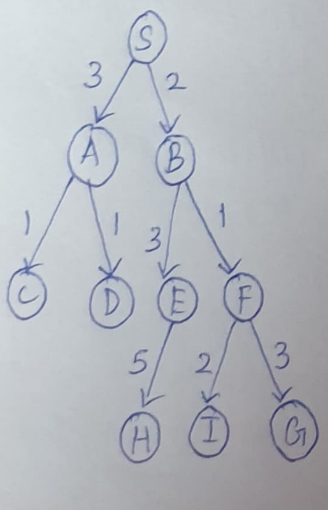
2. Explain in brief along with example following types of Informed Search Algorithms

- Best First search Algorithm (Greedy search)
- A* Search Algorithm
- AO* Search Algorithm

- <u>Best - First search Algorithm (Greedy search)</u>

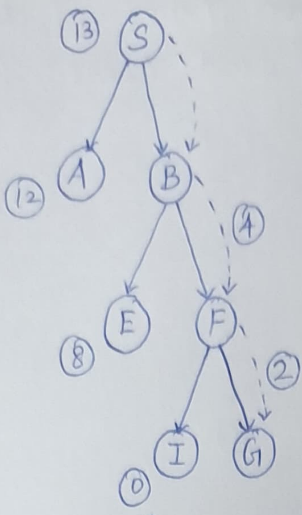
- Greedy search Algorithm always select path which appears best at the moment.
- It is the combination of depth fast search and breadth-first search algorithm.
- It uses the heuristic function and search
- Best-first search allows us to take the advantage of both algorithms
- with the help of best-first search, at each step, we can choose the most promising node

Eg: consider the below search problem, and we will traverse it best-first search. At each iteration, each node is expanded evaluation function $f(n) = h(n)$, which is given below



node	H(n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In the search example, we use two lists which are open and closed lists. Following are the iteration for traversing.



Expand the nodes of S and put in the CLOSED list

Initialization : open[A,B], closed[S]

Iteration 1 : Open [A], closed [S,B]

Iteration 2 : 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]

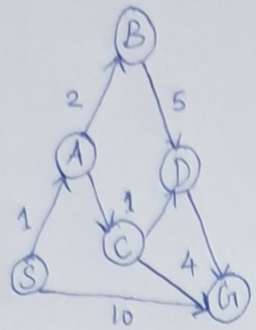
open[I,E,A], closed[S,B,F,G]

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

- A* search Algorithm
- A* search Algorithm 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 UCS and greedy best-first search, by which it solve the problem efficiency.
- 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.

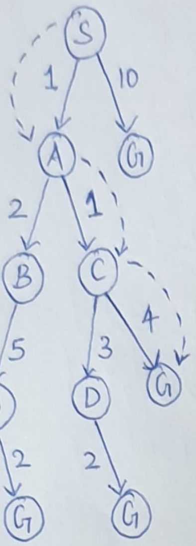
Eg: In this example, we will traverse the given graph using the A* Algorithm. The heuristic value of all states is given in the below table so will calculate the f(n) of each state using the formula $f(n) = g(n) + h(n)$, where g(n) is the cost

to reach any node from start state
Here we will use OPEN and CLOSED list



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

solution



Initialization : $\{(S,5)\}$

Iteration 1 : $\{(S \rightarrow A, 4), (S \rightarrow G, 10)\}$

Iteration 2 : $\{(S \rightarrow A \rightarrow C, 4), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 3 : $\{(S \rightarrow A \rightarrow C \rightarrow G, 6), (S \rightarrow A \rightarrow C \rightarrow D, 11), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 4 : will give the final result, as $S \rightarrow A \rightarrow C \rightarrow G$ it provides the optimal path with cost 6

- <u>AO* Search Algorithm</u>

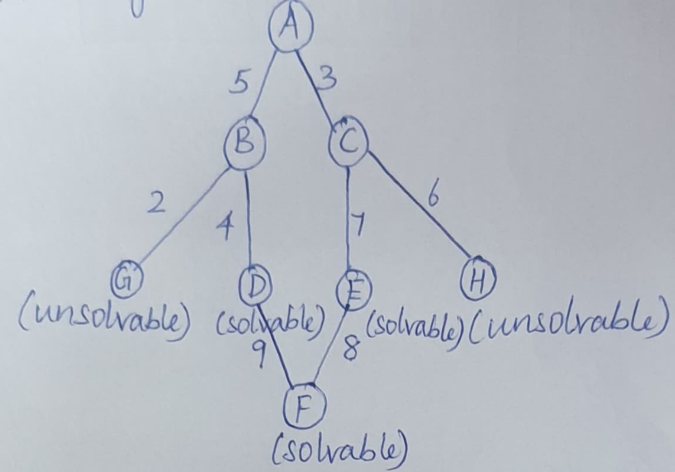
The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination condition are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do.
So for this purpose we are using AO* Algorithm.

Like A* Algorithm here we will use two arrays and one heuristic function.

OPEN: It contains the nodes that has been traversed but yet not been markable solvable or unsolvable

CLOSE: It contains the nodes that have already been processed.

Eg: Let us take the following example to implement the AO* Algorithm

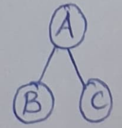


Step 1: In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.

i.e open = $\boxed{A}$ close = (NULL) $\boxed{\Phi}$ A

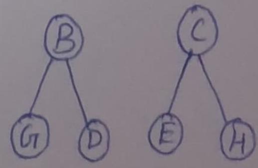
Step 2: The children of A are B and C which are solvable. So place into open and place A into the CLOSE

i.e OPEN = | B | C | CLOSE = | A |



Step 3: Now process the nodes B and C. The children of B and C are to placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

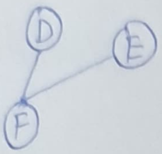
So OPEN = | G | D | E | | CLOSE = | A | B | C |



'0' indicated the nodes G and H are unsolvable

Step 4: As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e OPEN =



CLOSE =

A	B C	G(0)	D E	H(0)

Step 5: Now we have been reached at our goal place. so place F into CLOSE.

A	B C		G(0)	D E	H(0)	F

i.e CLOSE =

Step 6:

success and Exit

AO* Graph