Artificial Intelligence (Basics)

search - process of finding goal state with rules and combination of control strategy.

search algorithms :

- uninformed search (blind)
- informed search (Heuristic).

uninformed search

- no prior knowledge (domain specific knowledge).
- systematic exploration
- can be inefficient
- simple to implement
- BFS, DFS, DLS, IDS, UCS, bidirectional

Informed search / Heuristic search

- domain specific knowledge
- efficiency
- Heuristic fn is used (rule of thumb)
- quality of Heuristic fn impacts performance.

search data structures :

1) trees. nodes are connected by edges.
2) graphs (cyclic) node connected by edges.
3) queues (FIFO)
4) stacks (LIFO)
5) priority queues (elmt=priority)

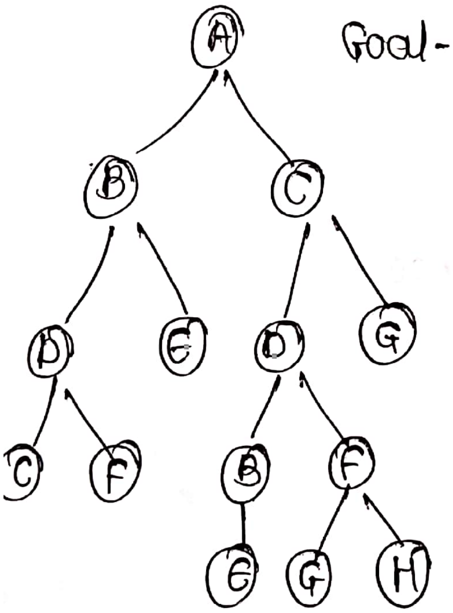
uniformed search

① breadth first search (BFS)

- highest layer of tree is searched then next layer.
- optimal soln (shortest path)
- not for large search space.
- time complexity $O(b^d)$
- uses queue (fifo)

O - order of
b - branch factor
d - depth.

Algo :
```
1. enter starting node in queue.
2. if queue is empty, return fail and stop.
3. if first elmt is goal, then return success and stop.
4. else remove first elmt and place its child nodes
   at end of queue.
5. Go to step 2.
```



Goal - G .

A		[ ]
check A	→	[B, C]
check B		[C, D, E]
check C		[D, E, D, G]
check D		[E, D, G, C, F]
check E		[D, G, E, F]
check D		[G, E, F, B, F]

check G = G ✓

travel path: A - B - C - D - E - D - G .

Goal path : A - C - G (parents pointers)

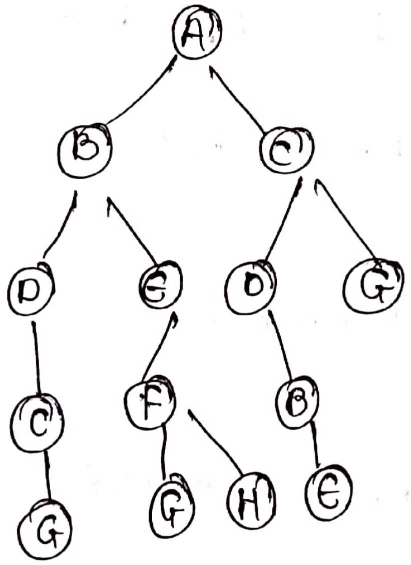
- graph traversal, network routing, pattern maching, data mining, games, web crawling, GPS navigation.

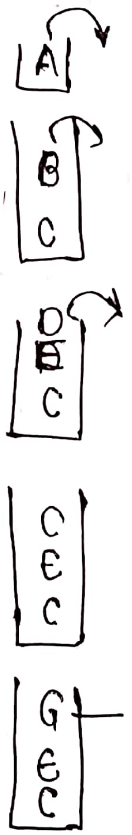
② Depth first search (DFS)

- extends current path as far as possible before backtracking to last choice point.
- no optimal soln guarantee
- optimal when search space is large.
- time complexity O(bm)

Algo:

```
1. put the start node in the stack.
2. while (stack is not empty)
     a. pop a node from the stack
          - if the node is goal state return success.
          - push all children of node into the stack.
3. return failure.
```





node visited {A}

LIFO N.V {A, B}

path - A-B-D-C-G

Goal state G = G

Advantage — low storage req; nodes ~~of~~ on current path are stored.
- optimal soln may earlier.

disadv - incomplete: without depth bound may not found soln

- maze solving, web crawling, cycle detection, scheduling problems, solving puzzles.

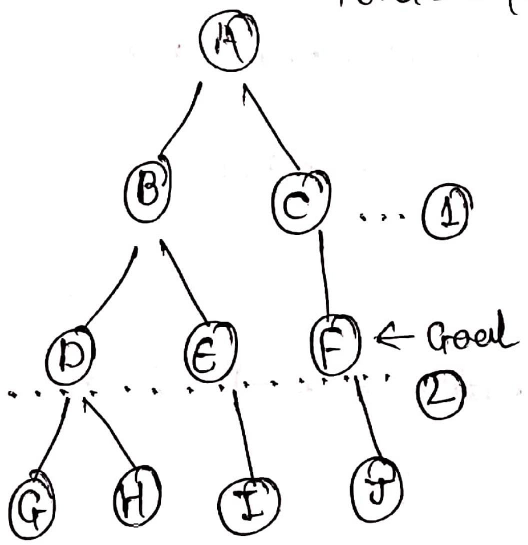
③ Depth limited search (DLS)

- introduce a depth limit on branches to be expanded
- ~~DLS~~ useful when we know the max depth of the soln.
- $O(b^l)$

advantage - memory efficient
- solves infinite path problem.

disadvantage - incompletness (if soln below limit)
- not optimal (many soln).

level=2 (depth limit).



stack [A] n.v {A} ~~B, D, E~~

[B C] n.v {A, B}

[D E C] n.v {A, B, D)

[E C] n.v {A, B, D, E}

[C] n.v {A, B, D, E, C}

[F] traversal path.

{A-B-D-E-C-F}

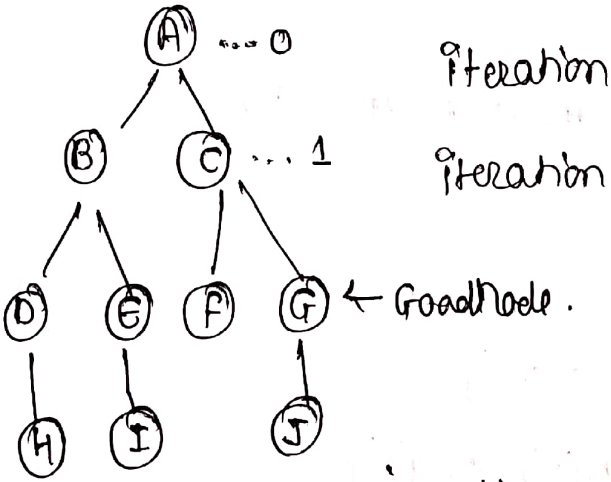
④ iterative deepening search (IDS)

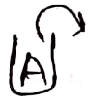
- gradually increase the limit 0, 1, 2, 3 ....
- combine DFS and BFS benefit.
- time complexity O[b^d]
- preferred when soln is unknown and space is large.

advantage: - completness
- find shallowest soln
- avoid infinite paths.

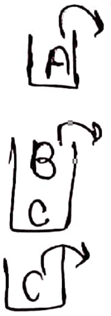
disadvantage - repeats all the work of peerious phase.
- time complexity slower.

- Game playing, pathfinding, AI planning



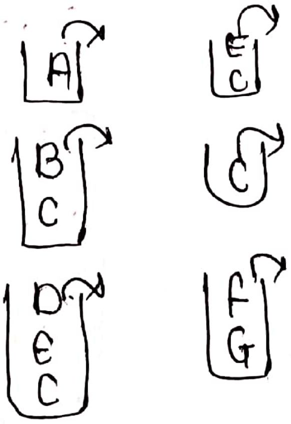
iteration 0 = 

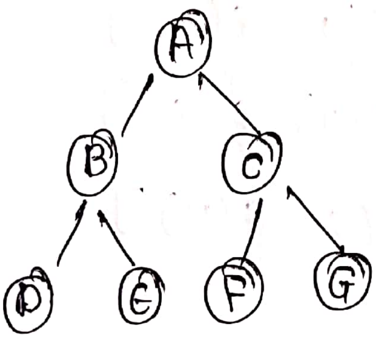
n.v {A}.

iteration 1 = 

n.v {A, B, C}.

iteration 2 = ↓





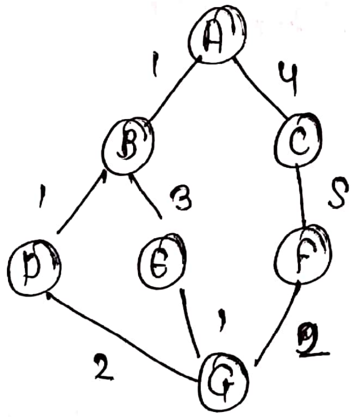
n.v{ A, B, D, E, C
F }.

⑤ uniform cost search (UCS) or branch and bound.

- based on cost associated with path.
- sum of costs is least - path
- priority queue.
- time complexity $O(b^{1+c/\epsilon})$

Algo - 1. insert root node into priority queue.
2. remove element with highest priority.
3. if removed node is goal node
   - print total cost and stop the algorithm
4. else
   - enqueue all children to priority queue, with cumulative cost from the root as priority and the current node to the visited list.

adv - completeness and optimal
dis - may in infinite loop (concerned about path cost)



[A]

[B(1) C(4)]

[C(4) D(2) E(4)]

[D(2) E(4) F(9)]

[E(4) F(9) G(4)]

[F(9) G(4) G(5)]

[$\underline{G(4)}$ G(5) G(11)]
✓

Shortest path - A - B - D - G
cummulative cost - 4

6 Bidirectional search

- 2 searches from initial state and goal state
- goal state is replace with 2 search intersect.
- time complexity $O[b^{d/2}]$

Algo : 1. A is initial node and O is goal node and H is intersection node.

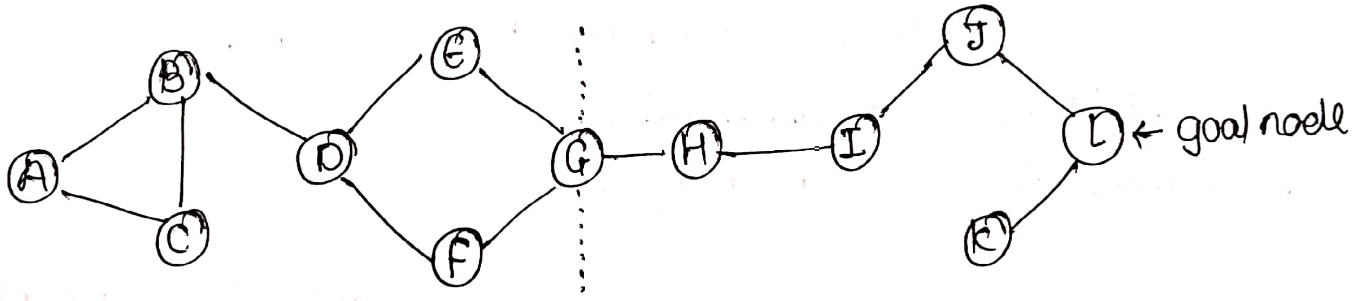
2. We will start searching simultaneously from start to goal node and backward from goal to start node.

3. Whenever the forward search and backward search intersect at one node, then searching stops.

Adv - fast and req less memory.

dis - more complex, difficult implementation.

- route planning, game playing, robotics.



node visited	queue.
A	[A]
A B	[B, C]
A B	[C, D]
A B C	[D]
A B C D	[E F]
A B C D E	[F G]
A B C D E F	[G]

A - B - D - E - G

node visited	queue.
L	[L]
L	[J K]
L J	[K, I]
L J K	[I]
L J K I	[H]
L J K I H	[G].

L - J - I - H - G

combine : A - B - D - E - G - H - I - J - L

Informed search Algorithms.

h(n) heuristic fn - calculate optimal path betn pair of states. (+)

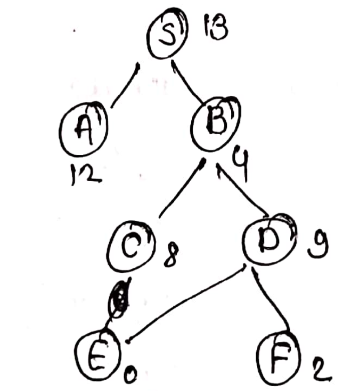
① **Best first search** (greedy Best first search)

- tree search based on f(n) expansion.
- low f(n) expand first.
- path finding, machine learning and optimization, network routing.

```
Algo : 1. Start   begin at initial node, add to priority queue
       2. loop: while p.queue is not empty.
            - dequeue: remove most promising node
                       from queue.            low heuristic
            - goalcheck: node is goal, done!
            - expand: otherwise expand node
            - enqueue: add successors to p.queue
                       ordered by promise (f(n)).
       3. no soln: if priority queue becomes empty.
                   there is no path to the goal.
```

✓ efficiency ✗ optimality

✓ flexibility ✗ memory.



(0 - goal state)

		remove node	parent node
1.	S(13)	-	—

2.	B(4)
	A(12)
		S	—

3.	C(8)
	D(9)
	A(12)
		B	S

4.	D(8)
	A(12)
		C	B

5.	E(0)
	F(2)
	A(12)
		D	B

6.	F(2)
	A(12)
		E	D

optimal path S-B-D-E

(2) A* search

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

  $g(n)$ — path cost.

  $h(n)$ — heiuristic value.

- optimal soln
- admissible $h(n) \leq h^*(n)$.

Algo : 1. intialization :

- start with initial node and add it the open list
- calculate cost fn $f(n) = g(n) + h(n)$.

2. iteration : While open list is not empty.

- select node with lowest $f(n)$
- remove from openlist ,add to closed alist.
- generate successors (neighbors) of the current node.
- for each successor
  - if the successor is not in the closed list.
    - calculate $g(n)$, $h(n)$, $f(n)$
    - add to open list.

3. Goal test -

- if the goal nod reached, backtrack parent pointers reconstruct shortest path.

✓ - faster than other uniformed (BFS and DFS)

✗ - stuck in loop

- not optimal. (unguided search in worst case)

- pathfinding, machine learning aptmization, network routing.