AIML Assignment.

Name: K. Jaswitha Reddy.

Roll no.: 2320030014

Sec: 04

Uninformed Searching Techniques:

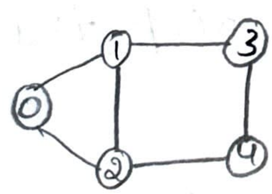
Breadth First Search (BFS):

Initialization:

* Start at a root node (or any orbitary node in the case of a graph).
* Use a queue (FIFO) to keep track of unvisited nodes.
* Mark the starting node as visited.

Example:

1.

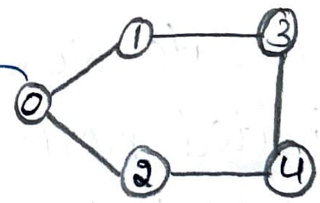


visited: | | | | | |

Queue: | | | | | |

Front.

2.

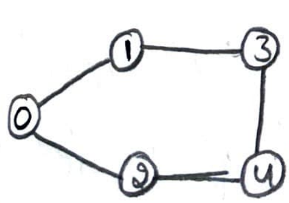


visited: | 0 | | | | |

Queue: | 0 | | | | |

Front

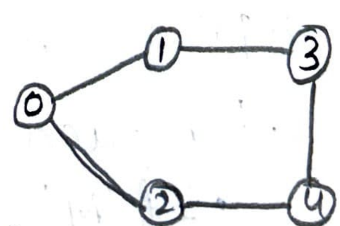
3.



visited: | 0 | 1 | 2 | | |

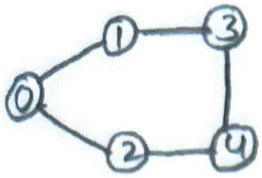
Queue: | 1 | 2 | | | |

4.



visited: | 0 | 1 | 2 | 3 | |

Queue: | 2 | 3 | | | |

5. 

visited: | 0 | 1 | 2 | 3 | 4 |

Queue: | 3 | 4 | | | |

↑Front.

6. visited: | 0 | 1 | 2 | 3 | 4 |

queue: | 3 | | | | |

↑Front

7. visited: | 0 | 1 | 2 | 3 | 4 |

Queue: | | | | | |

↑Front

Algorithm:

* Given a graph $G = (V, E)$ where;

  $V \rightarrow$ set of vertices

  $E \rightarrow$ set of edges.

and a starting node s.

* Create a queue 'Q' and add the starting node s to it.
* create set 'visited' to keep track of visited nodes.
* Add 's' to 'visited'
* While 'Q' is not empty.
  1. Dequeue a node 'N' in the front of queue
  2. process node 'N' (Print (or) store)
  3. For each neighbor (M) of node 'N'
     - If 'M' has not been visited
       - Enqueue 'M' to 'Q'
       - Add 'M' to visited.

Advantages:

* BFS will never get trapped exploring the useful path forever.
* Low storage requirement - linear with depth.

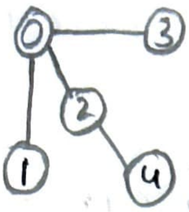
Applications:

* shortest path and minimum spanning tree for unweighted graph.
* Peer to peer networks.
* Broadcasting in network.
* Ford-Fulkerson algorithm.
* Image processing.

Depth First search (DFS):

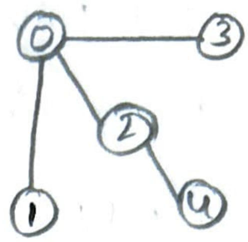
* Depth First search (DFS) is a recursive algorithm for searching all the vertices of a graph (or) tree data structure.
* It uses a stack to remember to get next vertex to get the next vertex to start a search.
* stack useus last in first out (LIFO) Principle.

Example:

1. 

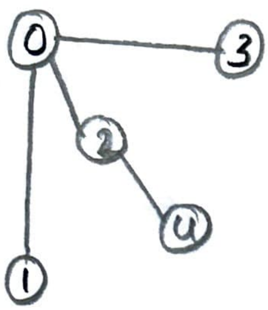
visited:

stack:

2. 

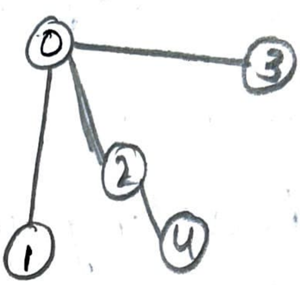
visited: 0

stack: 1 2 3

3. 

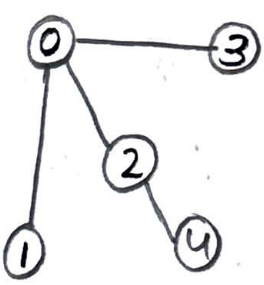
visited: 0 1

stack: 2 3

4. 

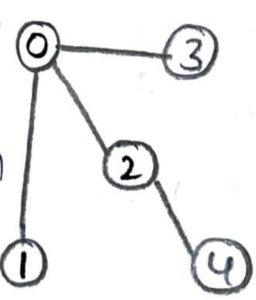
visited: 0 1 2

stack: 4 3

5. 

visited: 0 1 2 4

stack: 3

6. 

visited: 0 1 2 4 3

stack:

Algorithm:

* create a set 'visited' to keep track of visited nodes
* create a stack 's' and add the starting node 's' to it.
* Add s to 'visited.'
* while 's' is not empty:
    - pop a node N from the top of the stack.
    - Process the node (print (or) store).
    - For each neighbor M of node N;
        - If M has hot-been visited.
            - Push M onto the stack
            - Add M to visited
* The process continues until the stack is empty. At this point, all reacha-ble nodes from the starting node have been visited.

Applications:

* For finding the path.
* To test if the graph is biparticle.
* For finding strongly connected graph.

complexity:

Time: $O(V+E)$ $V \rightarrow$ no. of ~~edges~~ nodes

Space: $O(V)$ $E \rightarrow$ no. of edges

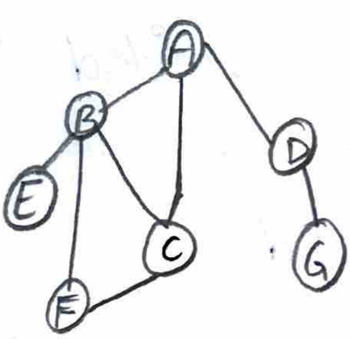
Iterative Deeping Search (IDS):

* combines the benefits of BFS and DFS by performing a series of depth-limited searches, each with an increasing depth limit

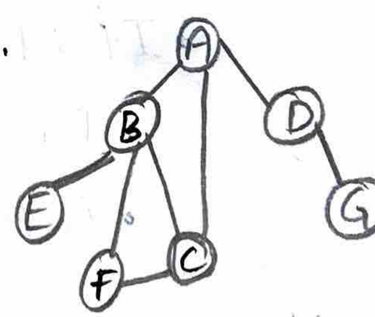
* Here, we do DFS in a BFS fashion (stack)

Example:

1.



2.



A → Source node.

D → solution node.

Here, deapth limit = O(L)

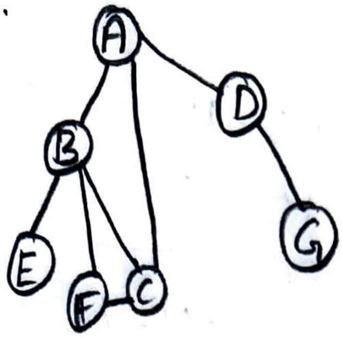
Place the reachable node in S_1 and mark visited.

A → S_1: A

Explore A, because the current level is already at the max. depth L. There will be no reachable nodes.

Takes A from S_1 & start the process again.

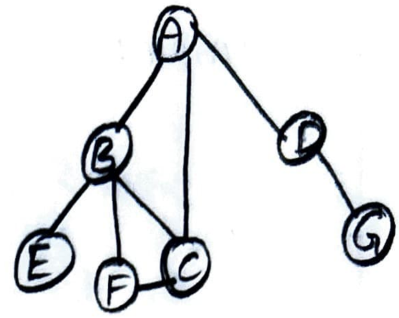
3.



Three nodes are accessible now B, C & D. Assume we start exploration with B.

B ⇒ pushed into S, and marked visited.

4.



After visiting C, D will be accessible from A, so D should be pushed into S, and marked visited

complexity:

Time: $O(b^d)$ b → branching factor

Space: $O(b \cdot d)$ d → shallowest node depth.

Informed Searching Techniques:

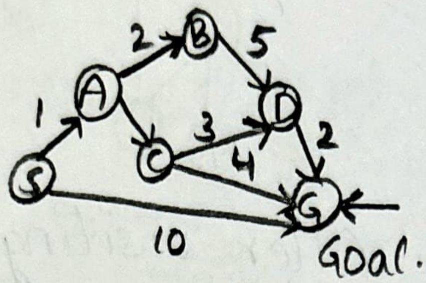
A^* Search:

* This informed search technique is also called as heuristic search.

* A^* uses $h(n)$ → heuristic fn. & $g(n)$ → cost to reach the node 'n' from start state.

* Estimated cost $f(n) = g(n) + h(n)$.

Example:



$S \to A = 1 + 3 = 4$

$S \to G = 10 + 0 = 10$

$S \to A \to B = 1 + 2 + 4 = 7$

$S \to A \to C = 1 + 2 + 1 = 4$

$S \to A \to C \to D = 1 + 1 + 3 + 6 = 11$

$S \to A \to C \to G = 1 + 1 + 4 = 6$

$S \to A \to B \to D = 1 + 2 + 5 + 6 = 14$

$S \to A \to C \to D \to G = 1 + 1 + 3 + 2 = 7$

$S \to A \to B \to D \to G = 1 + 2 + 5 + 2 = 10$

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

* A* Algorithm involves maintaining two lists:

- OPEN can have nodes that have been evaluated by the heuristic fn. but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

Algorithm:

* Define a list OPEN.
* Initially, OPEN consists soley of a single node, the start nodes.

* If the list is empty, return failure & exit.
* Remode 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.
* Expand node n.
* 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 the next step.
* For each successor node,
    - Apply the evaluation fn. f to the node.
    - If the node has not been in either list, add it to OPEN.
* Go back to step 3.

complexities:

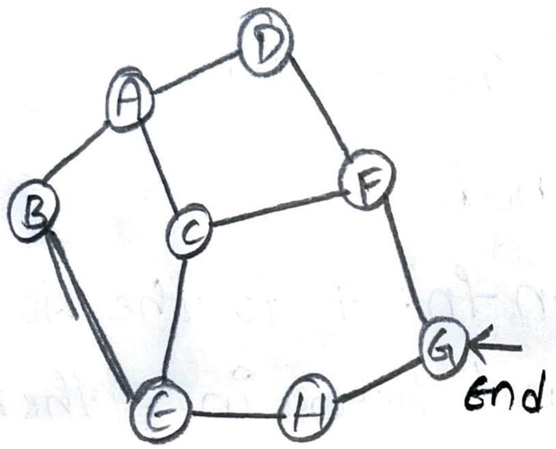
Time: $O(b^d)$ b → branching factor.

Space: $O(b^d)$ d → shallowest node depth.

Best first Search:

* This algorithm uses evaluation fn. to decide which adjacent node is most Promising and then explore.
* Priority queue is used to store cost of function.
* Implementation involves maintaining 2 lists: CLOSED & OPEN

Example:



st. line distance:

A → G = 40
B → G = 32
C → G = 25
D → G = 35
E → G = 19
F → G = 17
H → G = 10
G → G = 0

OPEN	CLOSE
[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

Path: A → C → F → G

cost: 44

Algorithm:

* Priority queue 'PQ' containing initial states. loop
* If PQ = Empty, Return fail.
* Insert node into PQ (open list).
* Remove first(PQ) → NODE (close-list)
* If NODE → GOAL,
  Return path from initial state to NODE.
  Else
* Generate all successor of NODE & insert newly generated NODE into 'PQ' according to cost value.

END loop.

complexities:

Time: $O(b^d)$ b → branching factor

Space: $O(b^d)$ d → shallowest node depth.

Hill climbing search:

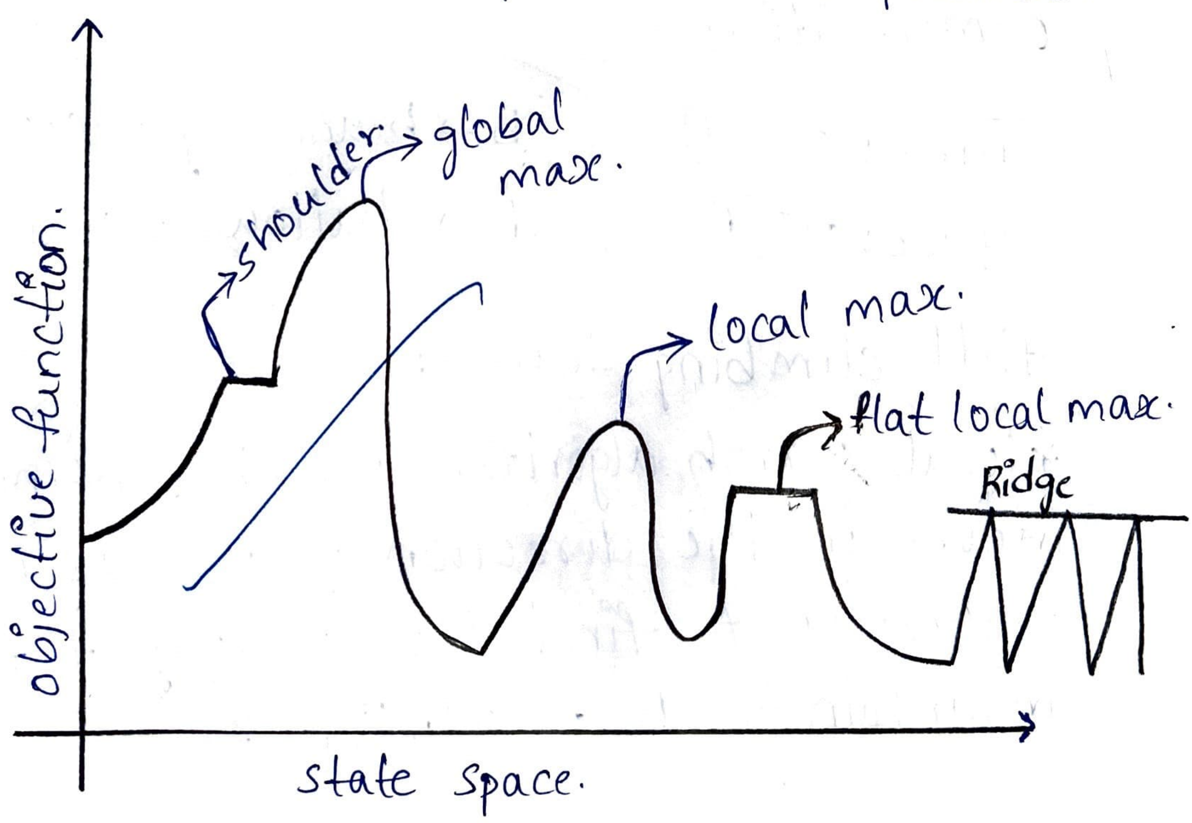
* local search algorithm which continuously moves in the direction of increasing evaluation to find the peak of the mountain (or) best solution to the problem.

* It is also called greedy local search. *
is mostly used when a good heuristic is available.

* simple Hill climbing.

Algorithm:

* Start an initial solution so.
* Repeat:
  - Generate a neighboring solution s' of the current solution s.
  - If f(s') (objective fn. value of s') is better than f(s)
    - Move to s' (i.e; set s = s')
  - Else keep the current solution
  - stop when no improvement is possible.



AL Assig

* steepest - Ascent hill climbing.

Algorithm:

* start with an initial solⁿ so
* Repeat:
  - Evaluate all neighbors of current solⁿ s.
  - choose the neighbor s' with best improvement (steepest ascent).
  - If f(s') is better than f(s):
    - More to s' (i.e; set s = s')
  - Else, stop as no better neighbor exists.
* stop when no improvement possible

Applications:

* Machine learning
* Robotics
* Natural language processing
* Network design.
* Data mining.

complexities:

Time: $O(d)$

Space: $O(b)$

$b \rightarrow$ branching factor.