Hill Climbing algo (Basic)

I/p → Problem

local variable → Current & neighbor

Current ← Make_node(Intial state(problem))

loop

do neighbor ← a highest_valued successor of current

If Value[neighbor] ≤ Value[current]

Then

Return State[current]

Current ← neighbor

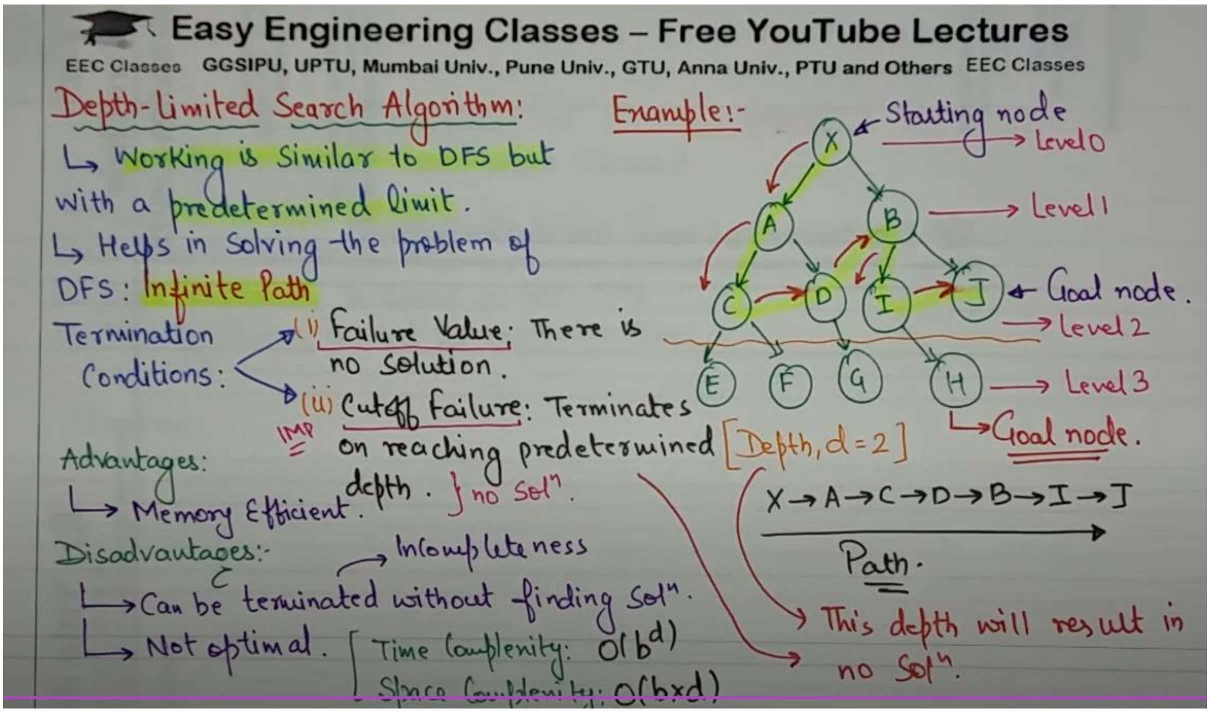
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

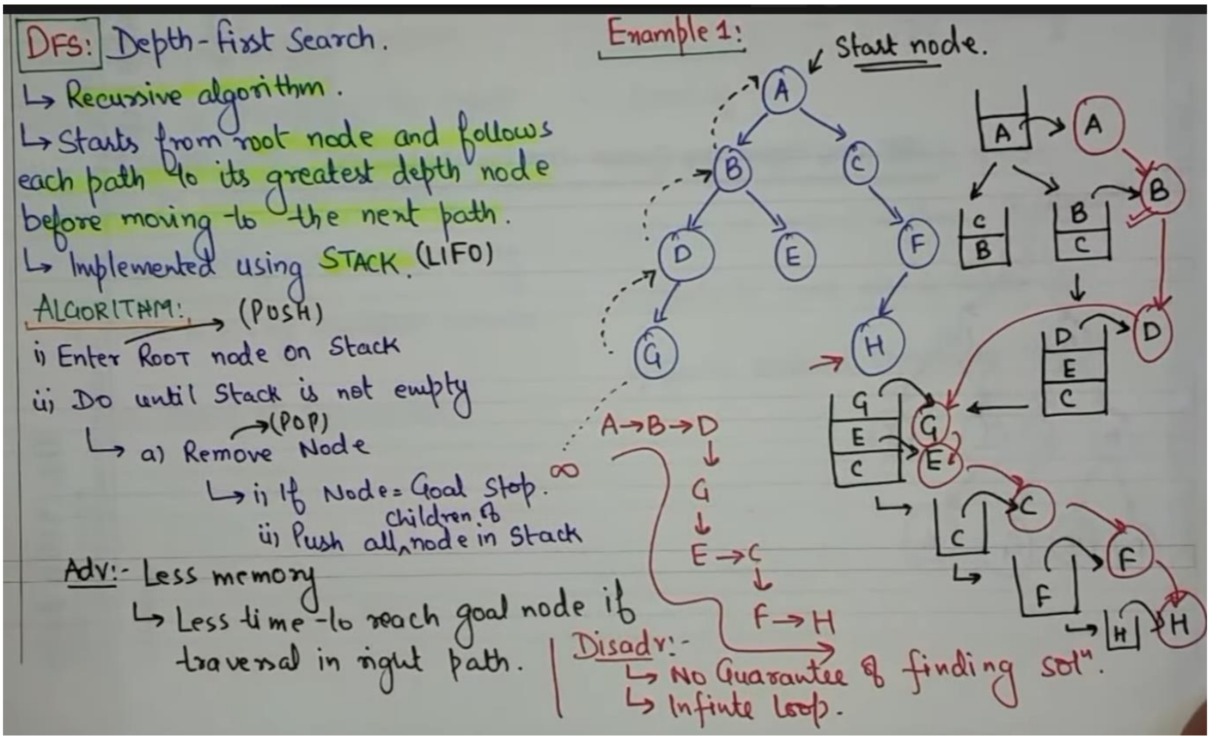
function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  current ← *problem*.INITIAL
  for t = 1 **to** ∞ **do**
    T ← *schedule*(t)
    if T = 0 **then return** *current*
    next ← a randomly selected successor of *current*
    ΔE ← VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{-\Delta E/T}$

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

Depth limit search



DFS



BFS

BFS:- Breadth First Search.

↳ Explores all the nodes at given depth before proceeding to the next Level.

↳ Uses Queue to implement.

ALGORITHM:

i) Enter Starting nodes on Queue.

ii) If Queue is empty, then return fail and stop.

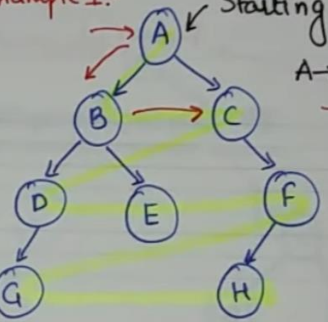
iii) If first element on Queue is GOAL Node, then return Success and Stop.

ELSE

iv) Remove and expand first element from Queue and place children at end of Queue. (IMP)

v) Goto Step (ii)

Example 1:-



A→B→C→D→E→F→G→H

Initial Queue {A} = Goal node ×

↳ Remove from Queue and add its successor to Queue.

{B, C} → (D, E)

{C, D, E}

{D, E, F} → {E, F, G}

{G, H}

Best first Search

BEST FIRST SEARCH: GREEDY SEARCH. (BFS DFS)

↳ Uses evaluation Algorithm (func^n) to decide which adjacent node is most promising and then explore.

↳ Category of Heuristic or Informed Search.

↳ Priority Queue is used to store Cost of nodes. → Sorted order.

ALGORITHM:

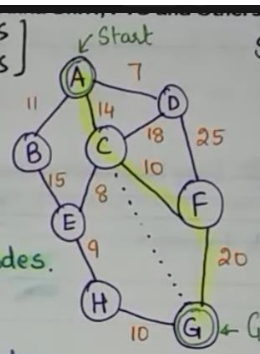
Priority Queue 'PQ' Containing initial States
Loop
if PQ = Empty Return FAIL
Else
NODE ← Remove_First(PQ)
if NODE = GOAL
Return Path from Initial to NODE
Else
Generate all Successors of NODE and insert newly generated NODE into PQ according to cost value.
END LOOP

Space Com:- $O(b^d)$

T.C:- $O(b^d)$

b: branching factor

d: depth



Straight Line distance.

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

Open	closed
[A]	[]
[C, B, D]	[A]
[B, D]	[A, C]
[F, E, B, D]	[A, C]
[G, E, B, D]	[A, C, F, G]

Path = A→C→F→G

(44)

A* algo(better version of best first search)

(A156)

Easy Engineering Classes – Free YouTube Lectures

EEC Classes GGSIPU, UPTU, Mumbai Univ., Pune Univ., GTU, Anna Univ., PTU and Others EEC Classes

Ques:- Explain A* Algorithm for Search.

↳ Uses heuristic func^n $h(n)$ and cost to reach the node 'n' from start state $g(n)$.

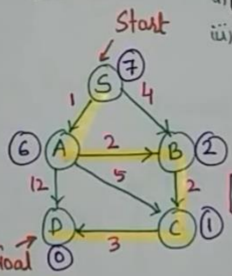
↳ Finds shortest path through search space.

↳ Fast and optimal result.

$$\underbrace{f(n)}_{\text{estimated cost}} = \underbrace{g(n)}_{\text{Cost to reach node}} + \overbrace{h(n)}^{\text{heuristic Value (child node)}}$$

ALGORITHM:

↳ i) Enter starting node in OPEN list.
ii) If OPEN List is empty return FAIL
iii) Select node from OPEN list which has smallest value (g+h).
↳ if node = Goal, return Success
iv) Expand node 'n' and generate all successors
↳ Compute (g+h) for each successor node.
v) if node 'n' is already in OPEN/CLOSED, attach to back pointer.
vi) Go to (iii)



Advantages:-

↳ i) Best searching algorithms.
ii) optimal and complete.
iii) Solving complex problems.

Disadvantages:-

↳ i) Doesn't always produces shortest.
ii) Complexity issues.
iii) Requires memory.

$S \rightarrow A = 1+6 = 7$ ✓
$S \rightarrow B = 4+2 = 6$ ✓
$S \rightarrow B \rightarrow C = 4+2+1 = 7$ ✓
$\boxed{S \rightarrow B \rightarrow C \rightarrow D = 4+2+3+0 = 9}$ r1
$S \rightarrow A \rightarrow B = 1+2+2 = 5$ ✓
$S \rightarrow A \rightarrow C = 1+5+1 = 7$ ✓
$\boxed{S \rightarrow A \rightarrow D = 1+12 = 13}$ r2
$S \rightarrow A \rightarrow B \rightarrow C = 1+2+2+1 = 6$ ✓
(#) $\boxed{S \rightarrow A \rightarrow B \rightarrow C \rightarrow D = 1+2+2+3 = 8}$ r3
$\boxed{S \rightarrow A \rightarrow C \rightarrow D = 1+5+3 = 9}$ r4

Problem formulation of zero sum or 2 player games

Two-player Games

- A game formulated as a search problem:

◦ Initial state:	board position and turn
◦ Operators:	definition of legal moves
◦ Terminal state:	conditions for when game is over
◦ Utility function:	a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA **payoff function**)

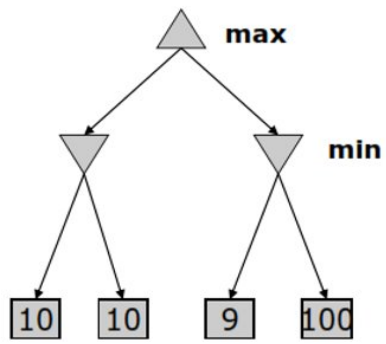
Min max algo

The minimax algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

The Minimax Algorithm Properties

- Performs a complete depth-first exploration of the game tree
- Optimal against a perfect player.
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$
- For chess, b ~ 35, m ~ 100
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
- Minimax serves as the basis for the mathematical analysis of games and for more practical algorithms



For alpha beta pruning the avg complexity would be b^m/2