{ O T X R S }

(11) { O T X R S }

(12) { ~~P q~~ } ⊙ ~~P T X~~ · { R S O T X }

(13) { ~~q r~~ } goal · { P T X }

(14) { q }.

⇒ Algo for Hill-Climbing:-

Step 1: Put the initial node on a list START.

2: If (START is empty) or (start = Goal) terminate search.

3. Remove the first node from start. Call this node a.

4. If (a = Goal) terminate search with succes.

5. Else if node a has successors, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from the goal and add them to the begining of start.

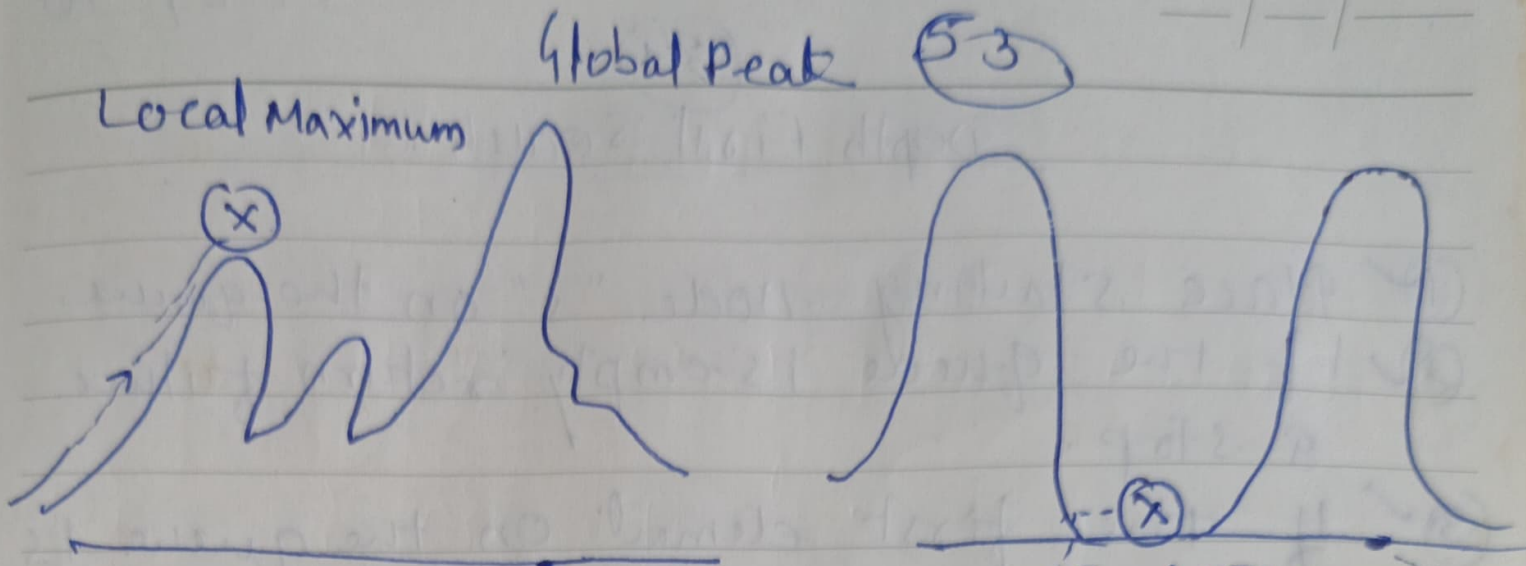
6 Goto step (4).

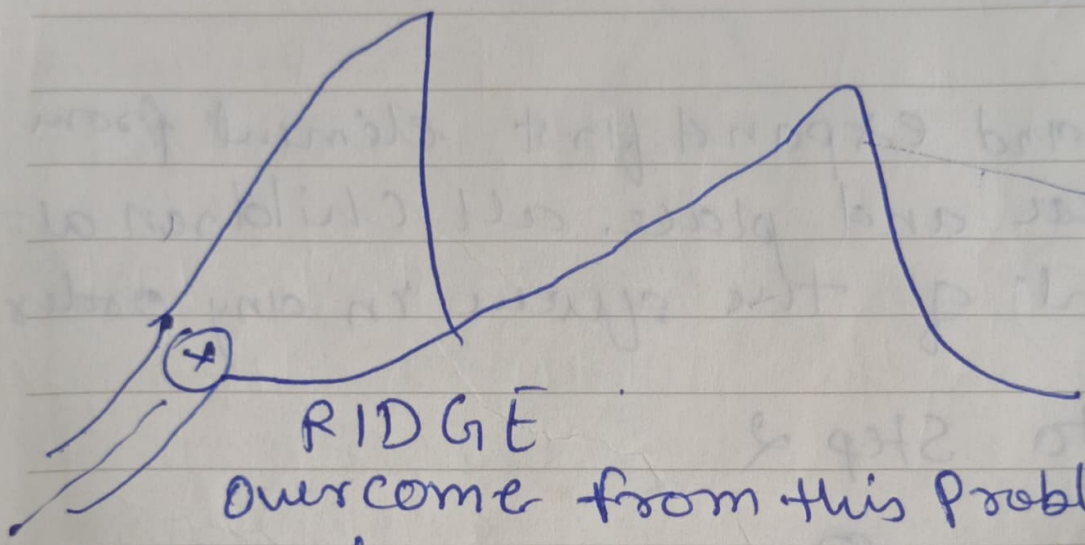
Problems of Hill-Climbing Technique:

i) Local Maximum: A state that is better than all its neighbours but not so when compared to states to states that are farther away.

ii) Plateau: A flat area of the search space in which all neighbours have the same value.

iii) Ridge: Described as "a long and narrow stretch of elevated ground or a narrow elevation or raised part running along or across a surface" by the oxford English Dictionary this is an area in the path which must be traversed very carefully as any movement in any direction might maintain one at the same level or result in fast descent.





Overcome from this Problems.

(i) Backtracking for local maximum. It helps in undoing what has been done so far and permits to try a totally diff path to attain the global peak.

(ii) A big jump is the solu^n to escape from the plateau. A huge jump is recommended bec^z in a plateau all neighboring pt^s have the same value

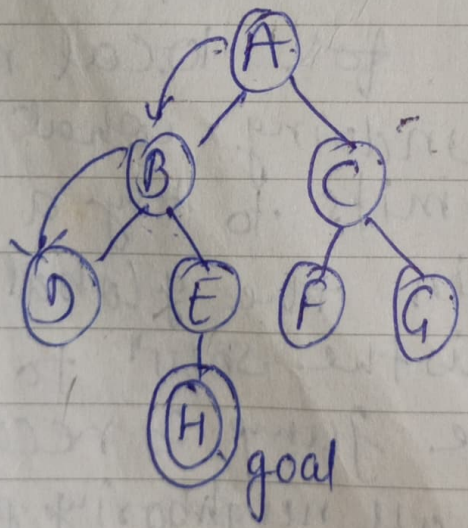
(iii) Trying diff^t paths at the same time is the solution for circumventing ridge.

21/07/10

(54)

Depth First Search.

(1) Place starting node "s" on the queue.
(2) If the queue is empty return failure & stop.
(3) If the first element on the queue is a goal node 'g' return success and stop. otherwise
(4) Remove and expand first element from the queue and place all children at the front of the queue in any order.
(5) Return to Step 2.

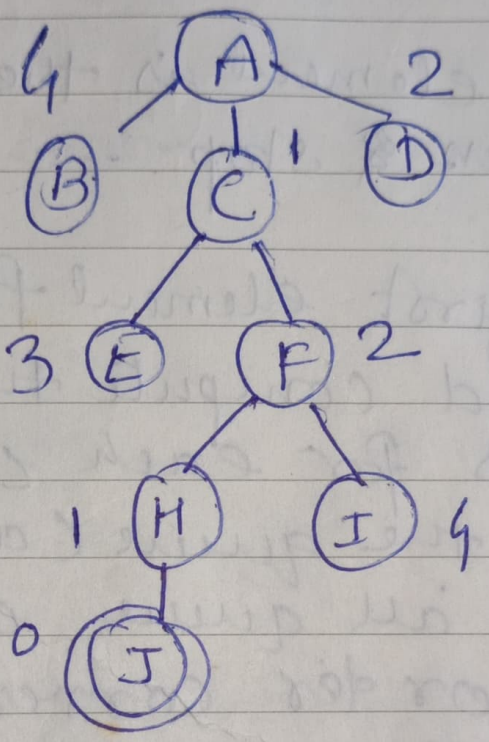


{A}
{BC}
{DEC}
{FGDE}
{EC}
{HC} success.

28x3 = 84

Informed Search :-

Hill Climbing Algorithm.



A → C → F → H → J

Best-First search

(1) Place the starting node 's' on the queue.

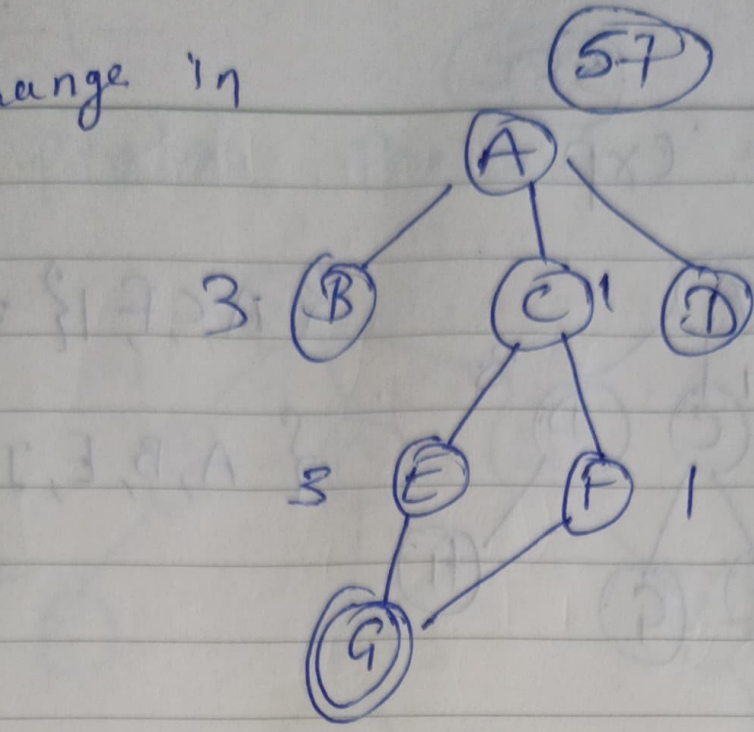
(2) If the queue is empty, return failure & stop.

(3) If the first element is the goal node 'g' return success & stop.
otherwise
Remove the first element from the queue expand it and compute the estimated goal distances for each child place the children on the queue (at either end) and arrange all queue elements in ascending order corresponding to goal distance from the front of the queue

(4) Return to Step 2.

B = 3, C = 1, D = 2.

arrange in



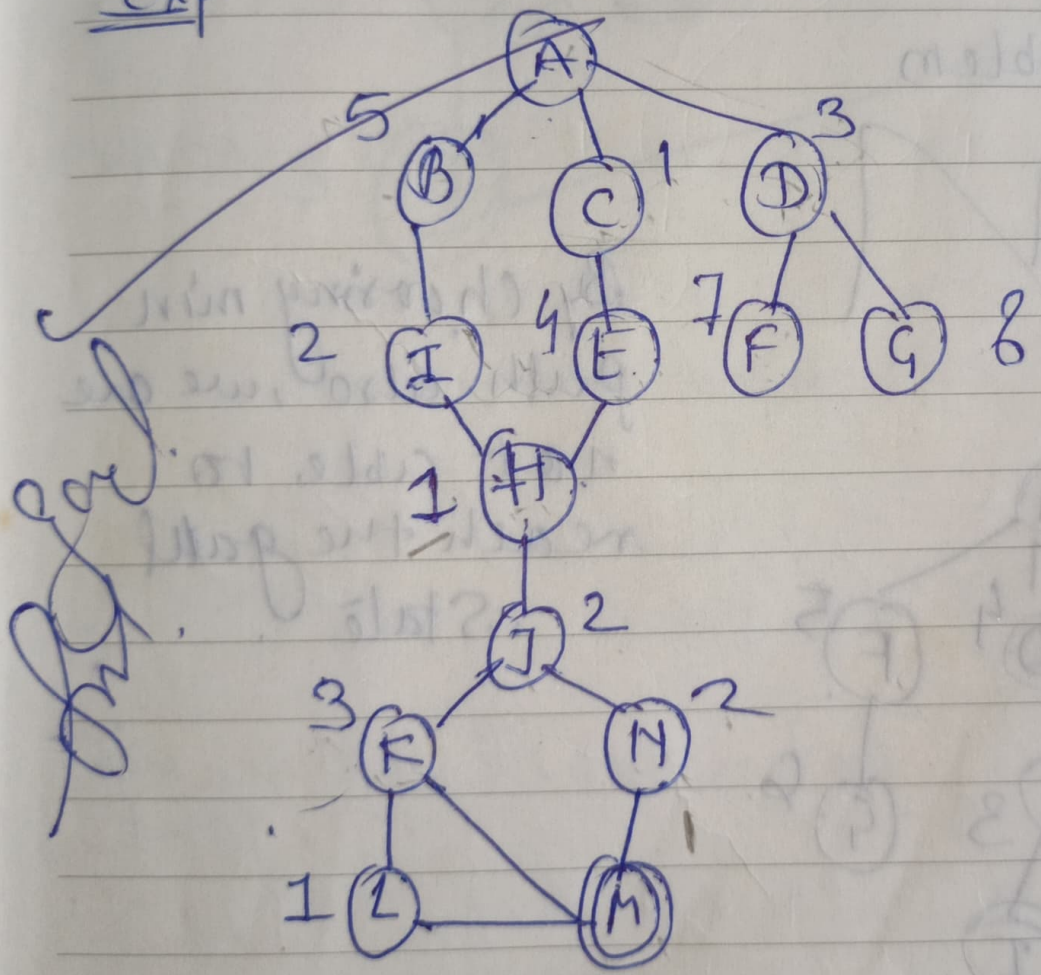
{A}

{CDB}

{FDEB}

{GDEB}.

Exp



{A}

{CDB}

{DEB}

{EBFG}

{HBFG}

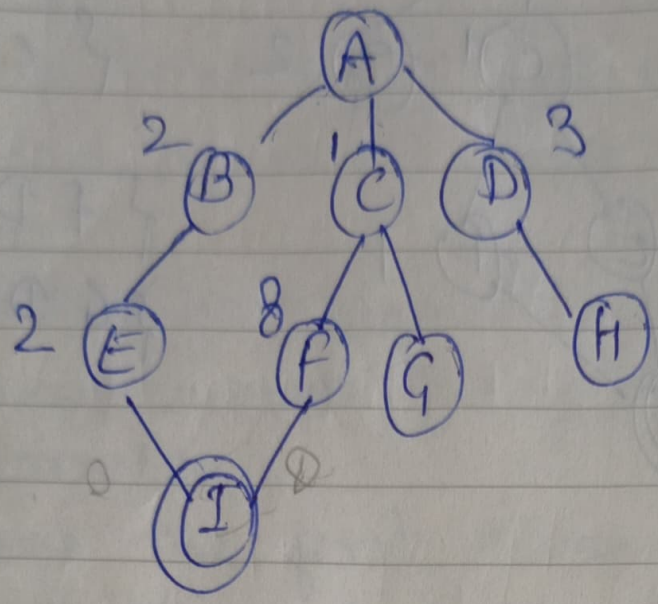
{JBFG}

{NkBFG}

{MkBFG}

goal.

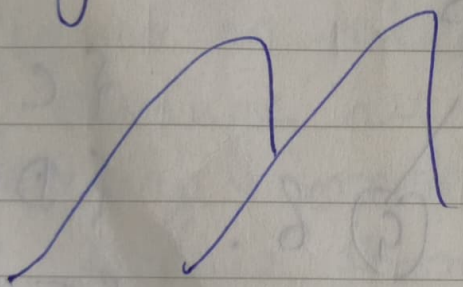
Local Maxima exp.



{A, C, F, I} = 9

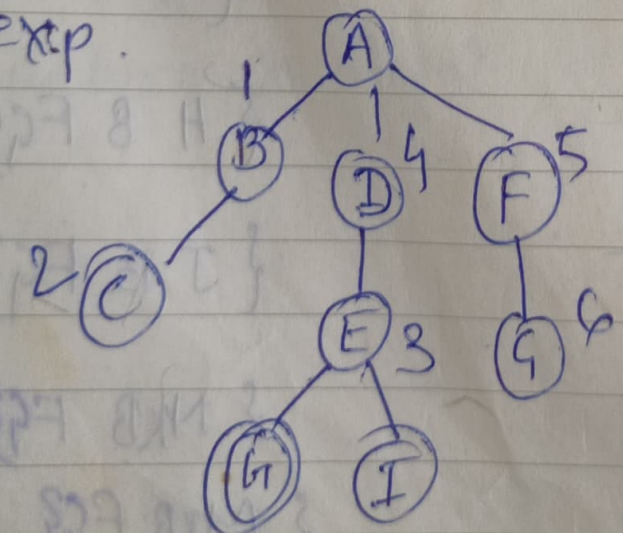
{A, B, E, I} = 4.

(2) Ridge Problem

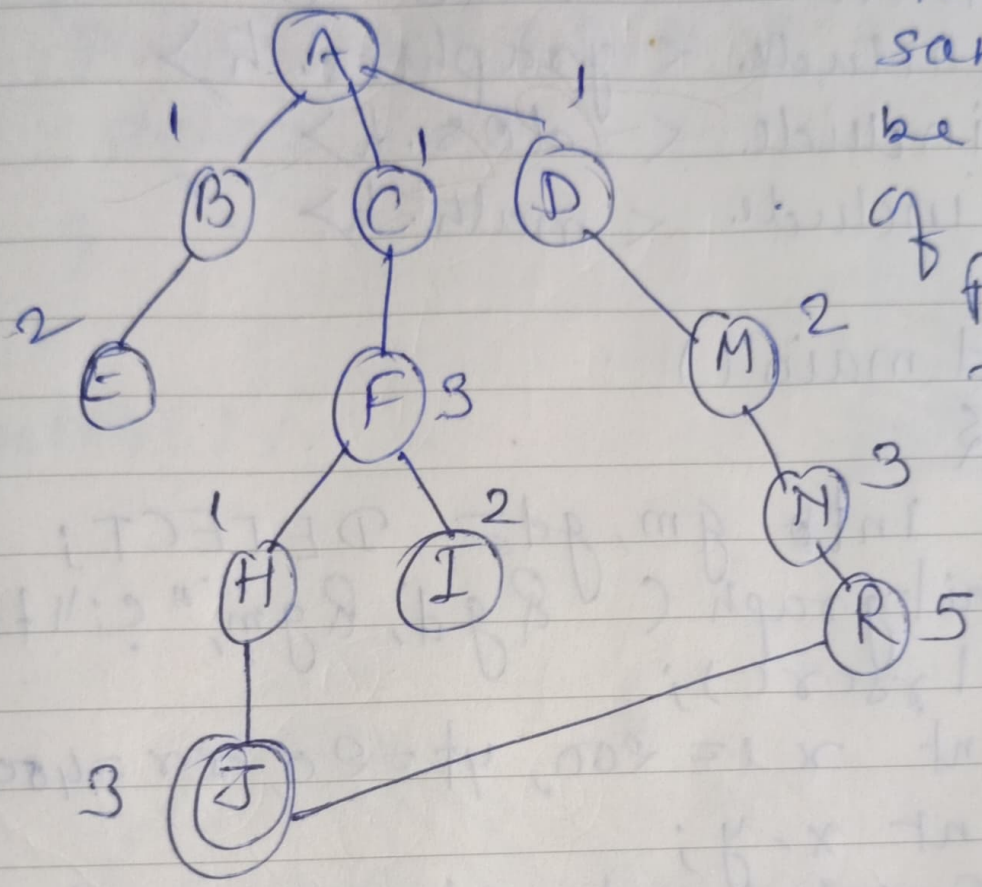


By choosing min path also, we are not able to reach the goal state.

exp.



(3) Plateau Problem

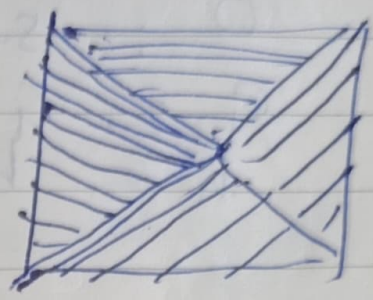


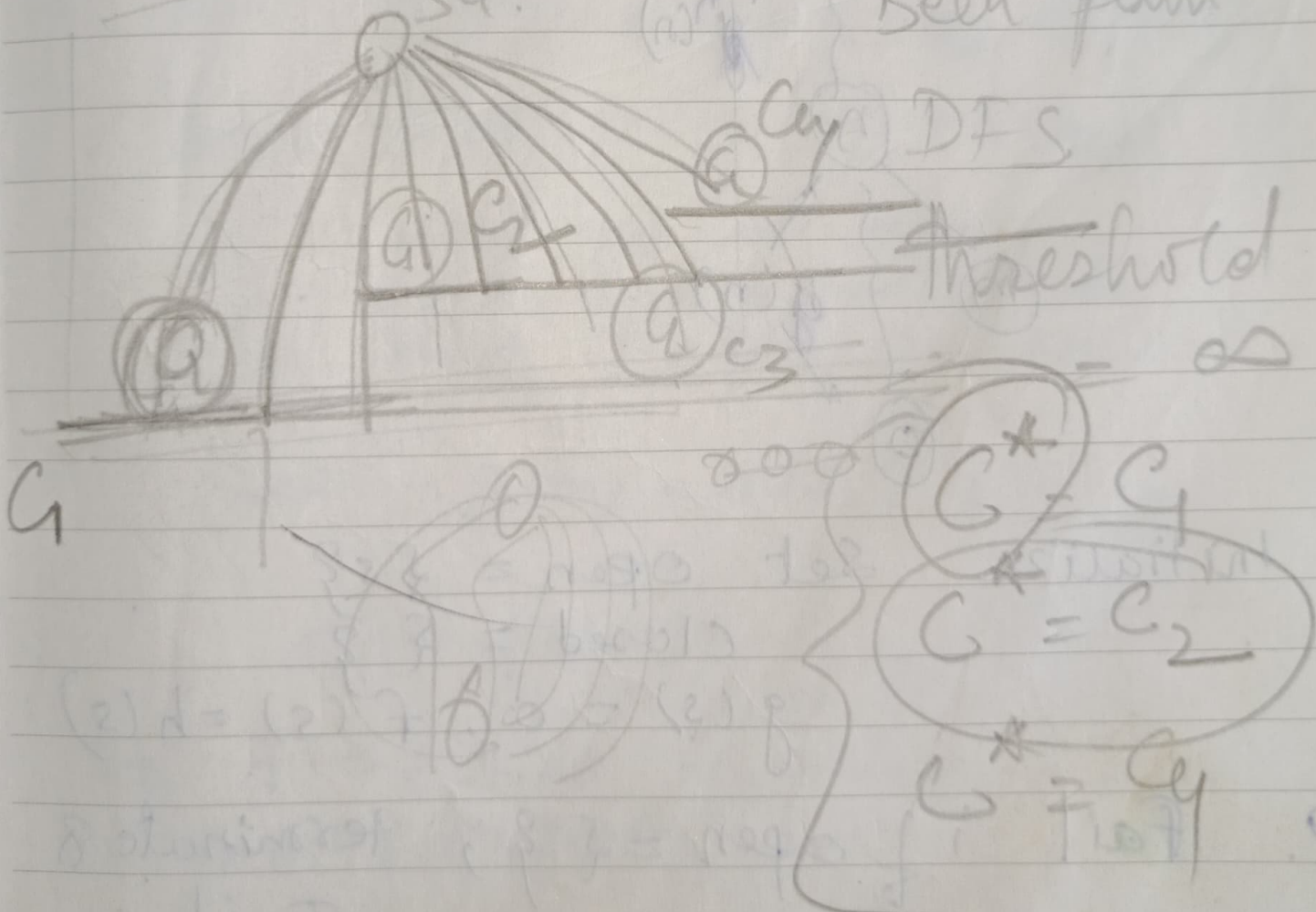
if the value of the path is same, there will be a confusion of choosing it. For solving this problem we come across Best first search.

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <dos.h>
#include <math.h>

void main()
{
    int gm, gd= DETECT;
initgraph( &gd, &gm, "c:\\tc\\bgi");
clrscr();
int xl=200, yt=20, xr=400, yb=400;
int x, y;
for (x=xl; xL=xr; x++)
        // loop for co-ordinates
{
for (y=yt; yL=yb; y++)
        // loop for y-co-ordinates.
{
if (xL=300) // to get the mid point.
{
putpixel(x, y, x+y);
delay(1); // Function to
                provide delay.
}
```

```
else
{
putpixel(x, y, x-y);
 delay(1);
}
}
}
getch();
}
```

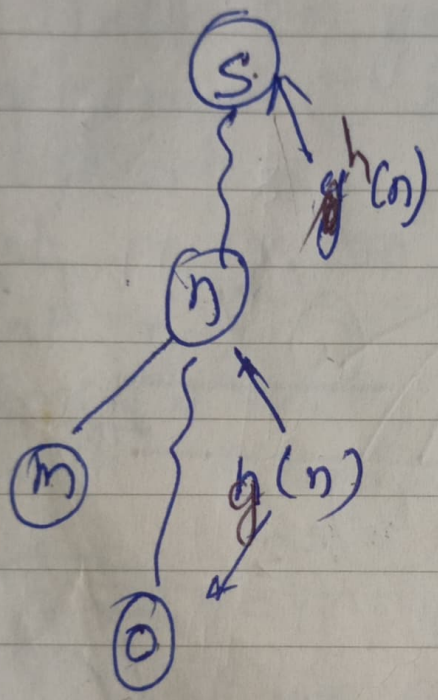




Algo for A*

Given : [S, s, O, G, h]

S: set of states space
s: start state
O : set of operators
G = Goal State (set of)
h = heuristic function.



1. Initialize Set open = {s}
   closed = { }
   g(s) = 0 . f(s) = h(s)

2. Fail : if open = { }, terminate & fail.

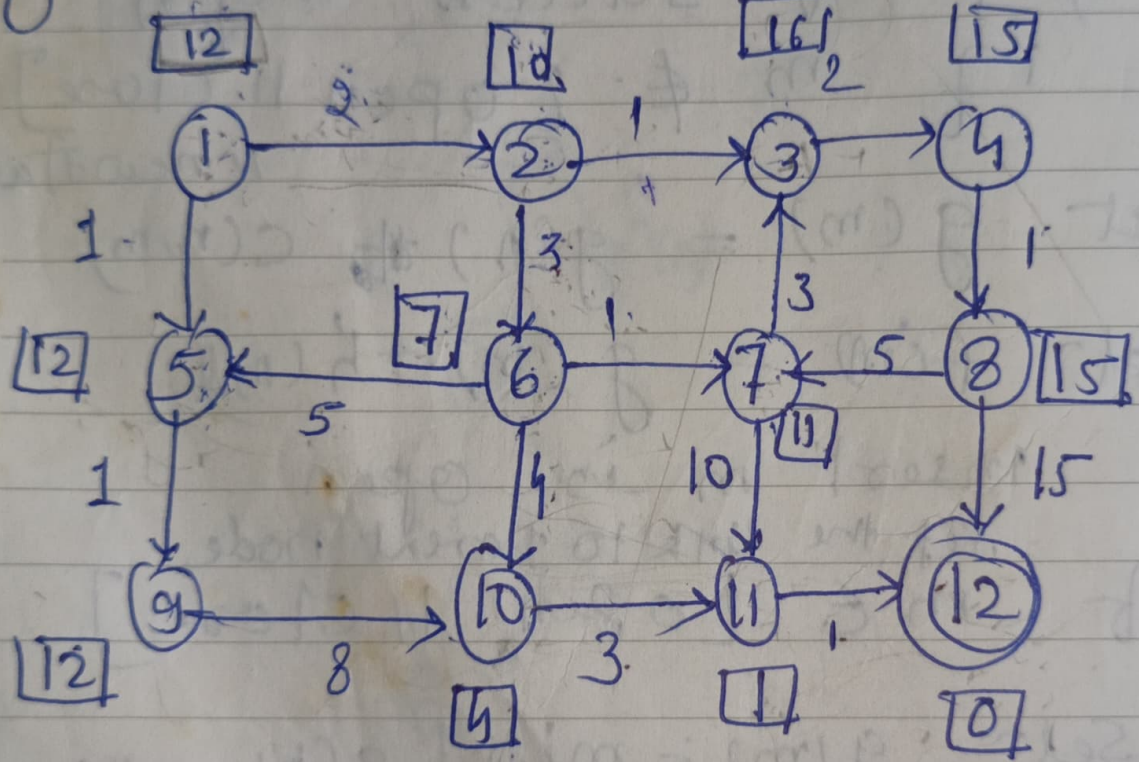
3. Select minimum cost state, n from OPEN. Save n in CLOSED.

4. Terminate :- If $n \in G$, terminate with success & $f'(n)$
↘ return ↗

5 Expand :- $\forall$ success m of n.
if $m \notin$ [open || close]
// new state
set $g(m) = g(n) + c(n,m)$
Set $f(m) = g(m) + h(m)$
Insert m in open
set the link to parent node
If $m \in$ [open || close]
Set $g(m) = \min [g(m), g(n) + c(n,m)]$
Set $f'(m) = g(m) + h(m)$
If $f(m)$ has decrease & $m \in$ {closed}.

move m to open
has decreased and
If f(m), is m∈open, update the cost
and change the link to parent node
6. Loop go to step 2.

Eg. $f(s) = g(s) + h(s)$. $0 + 12 = 12$ for 1

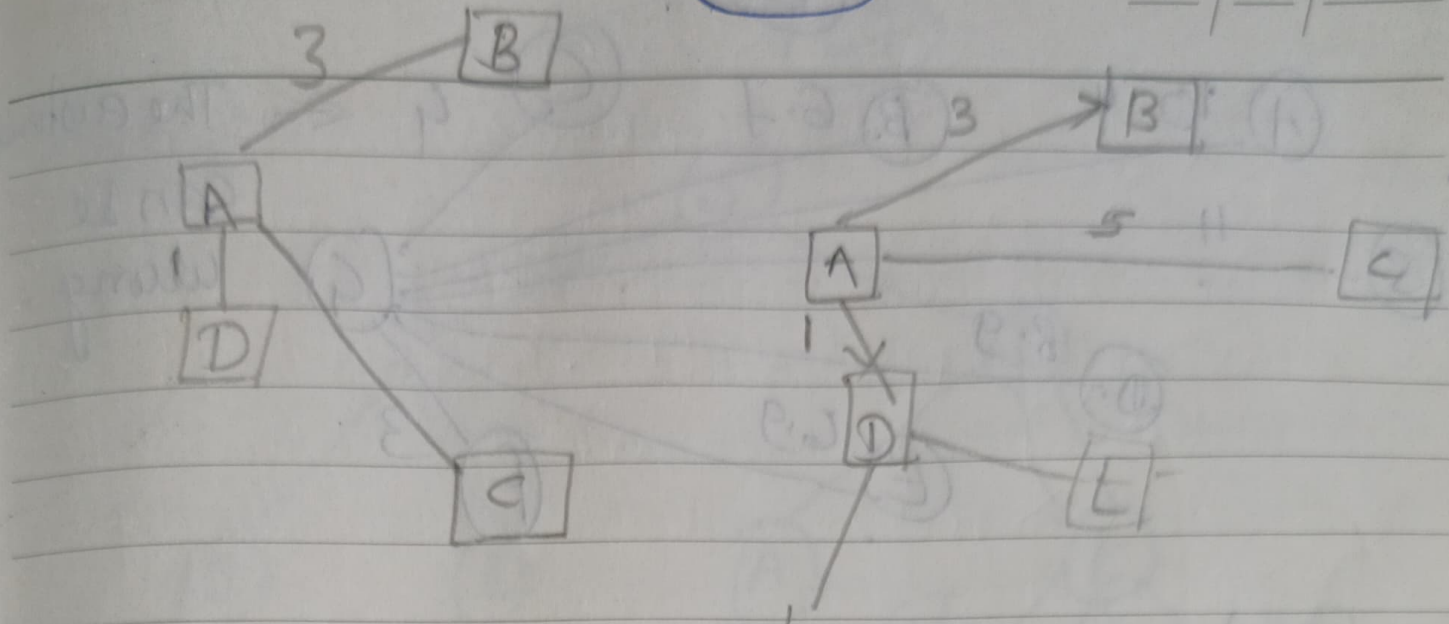


OPEN
1 12
2 12 5 13
5 13 3 19 6 12
5 13 3 19 7 17 10 13
3 19 7 17 10 13 9 14

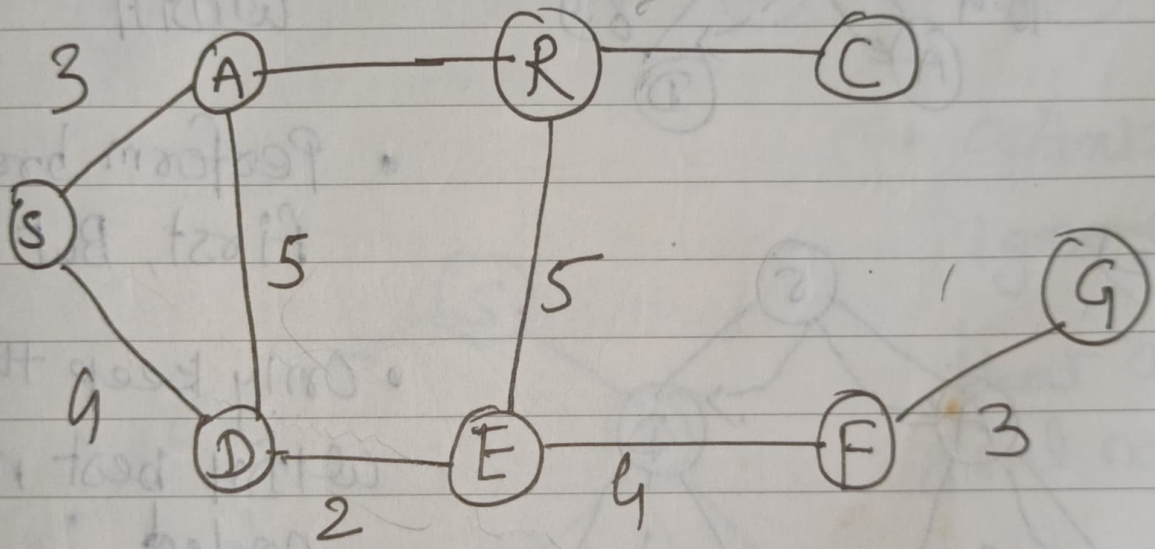
g = 0
f = h

3 19 7 17 9 14 11 13
3 19 7 17 9 14 11 13 12 13

Clare
1 12 2 12 6 12 5 13 10 13
11 13 12 13 goal
terminate.

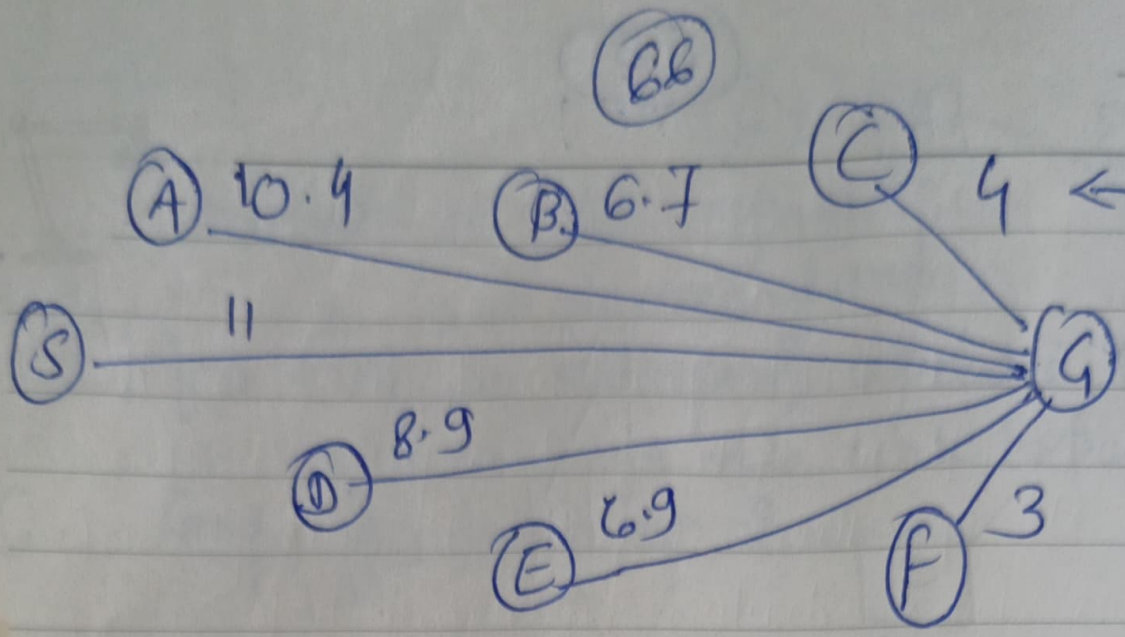


~~Branch & Bound~~ Beam Search.



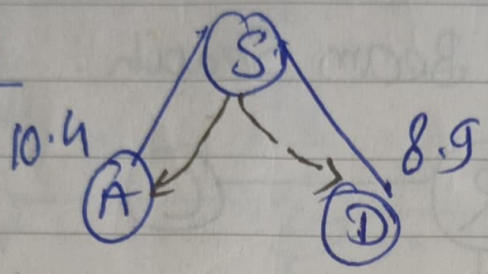
The Imagin the problem of finding a route on a road map and that the NET BULL is the road Map.

- Define f(T) = the straight line distance from T to G.



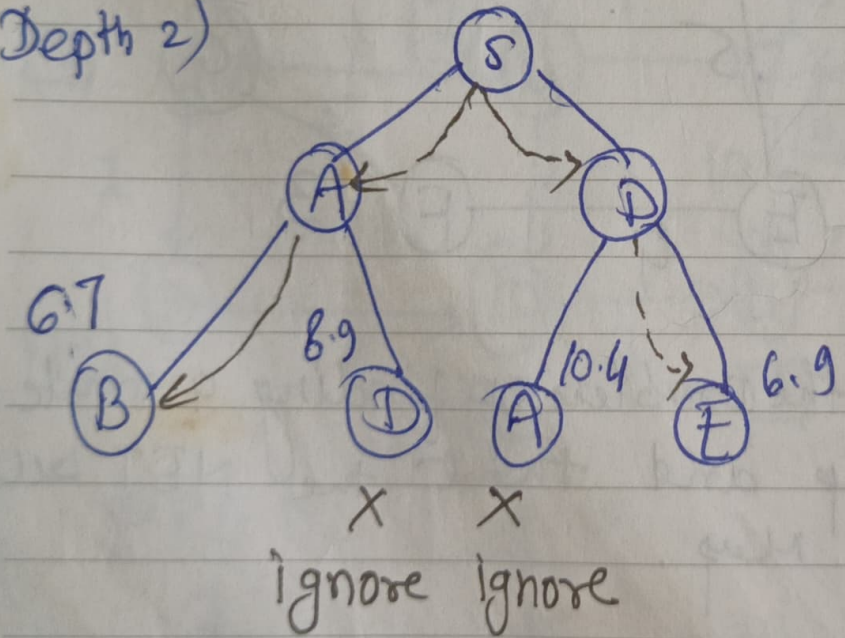
← The estimate can be wrong.

Depth 1)



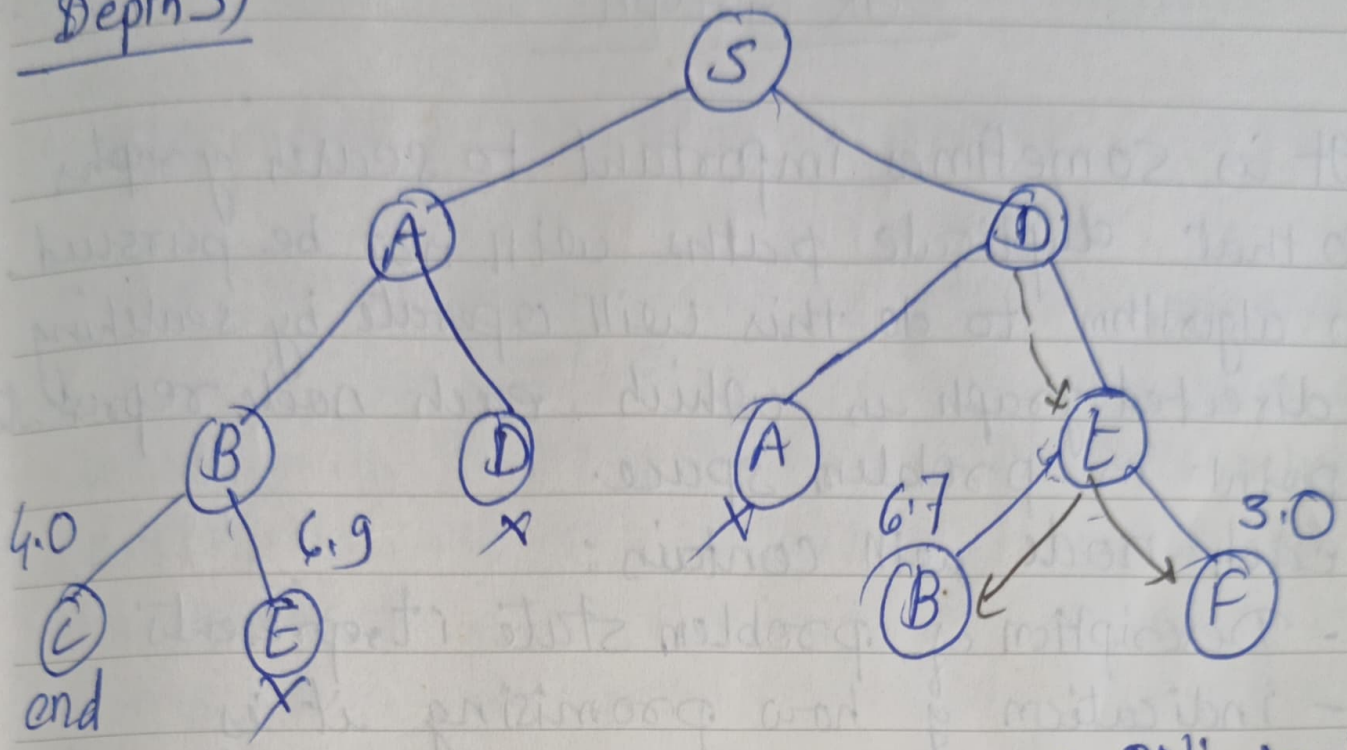
- Assume a prefixed WIDTH
- Perform breadth first, But:
- Only keep the WIDTH best new nodes depending on heuristic at each level.

Depth 2)



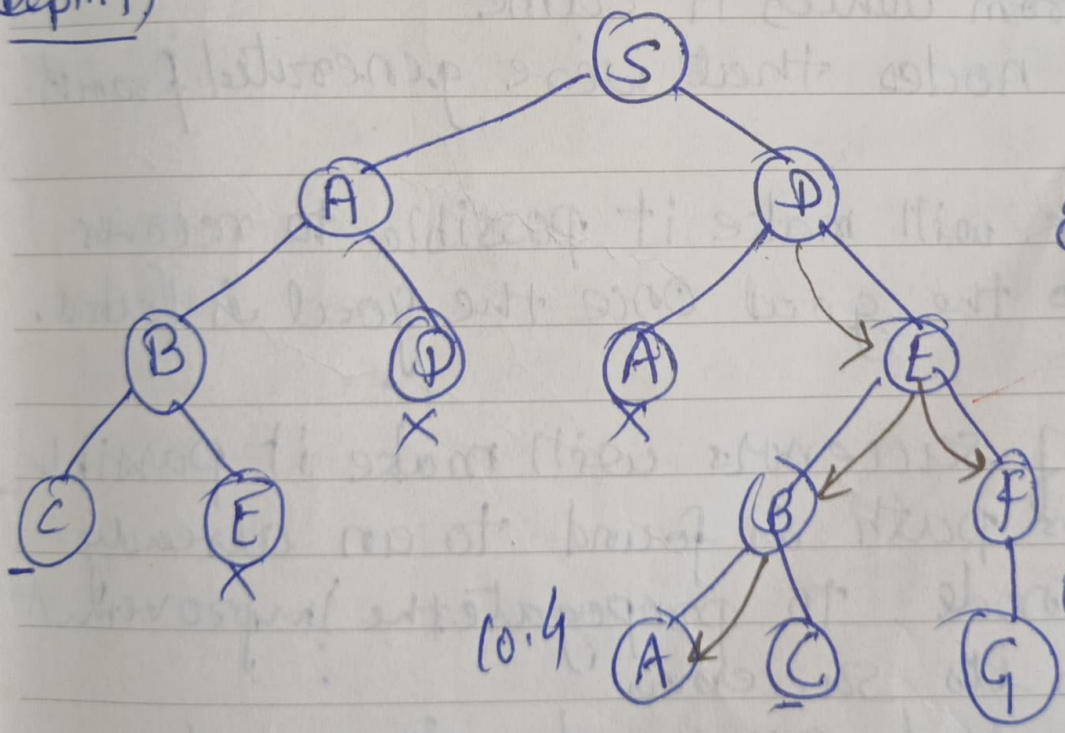
6.7

Depth 3)



- optimization: ignore leafs that are not goal nodes. (see e)

Depth 4)



OR Graph

- It is sometimes important to search graphs so that duplicate paths will not be pursued.
- An algorithm to do this will operate by searching a directed graph in which each node represents a point in problem space.
- Each node will contain:
  - Description of problem state it represents
  - Indication of how promising it is
  - Parent link that points back to the best node from which it came
  - List of nodes that were generated from it.
- Parent link will make it possible to recover the path to the goal once the goal is found.
- The list of successors will make it possible if a better path is found to an already existing node, to propogate the improvement down to its successors
- This is called OR-Graph, since each of its branches represents an alternative problem solving path.

Implementation of OR Graphs.

- we need two lists of nodes:

- OPEN - nodes that have been generated and have had the heuristic function applied to them but which not yet been examined. OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.

- CLOSED - nodes that have already been examined. we need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

A* Algorithm

- BFS is a simplification of A* Algorithm
- Presented by Hart et al.
- Algorithm uses:

- f': Heuristic function that estimates the merits of each node we generate. This is sum of two components, g and h' and f' represents an estimate of the cost of getting from the initial state to a goal state along with the path that generated the current node.

- g: The function g is a measure of the cost of getting from initial state to the current node.

- h': The function h' is an estimate of the additional cost of getting from the current node to a goal state.

- OPEN
- CLOSED.

Observations about A*

- Role of g function: This lets us choose which node to expand next on the basis of not only of how good the node itself looks but also on the basis of how good the path to the node was.
- h', the distance of a node to the goal. If h' is a perfect estimator of h, then A* will converge immediately to the goal with no search.

Exp 8 Puzzle Evaluation function.

$$f(n) = d(n) + w(n)$$

w(n) counts the number of displaced tiles in that data base associated with the node n

d(n) is the depth of the node in the search tree.

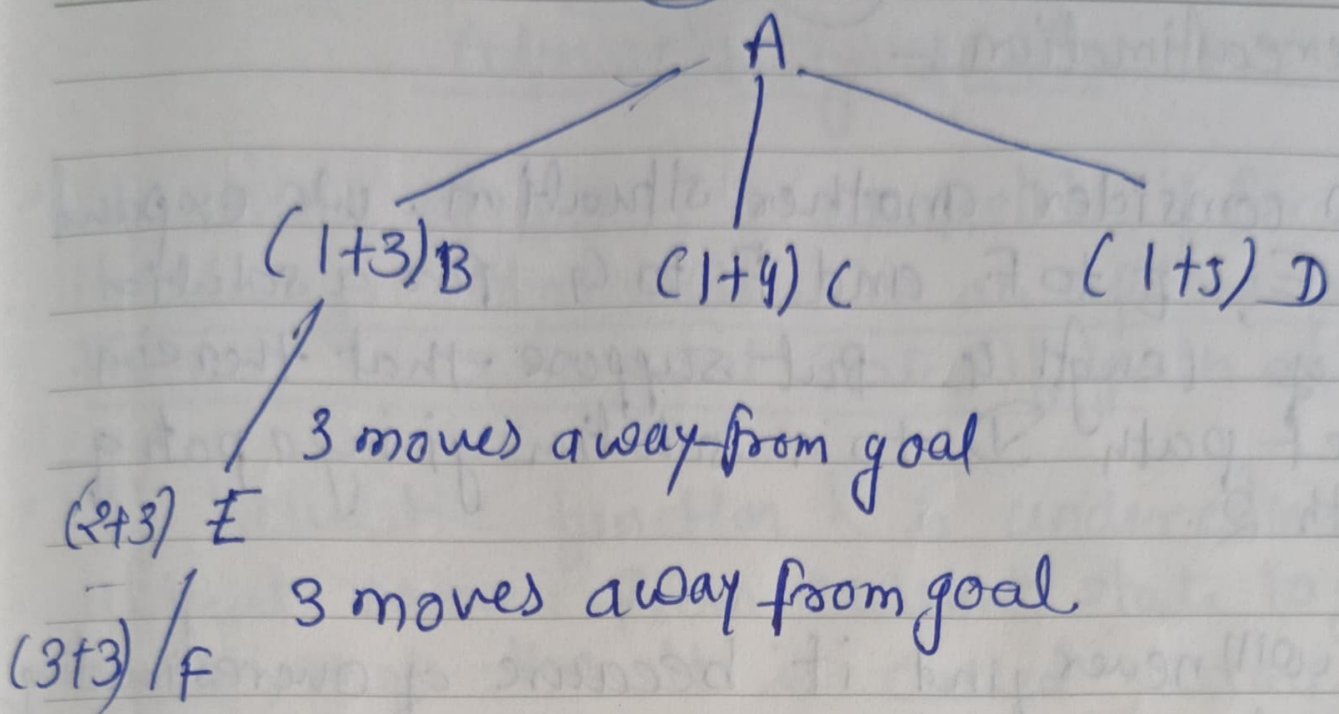
A* Special Cases.

- Suppose $h(n) = 0$. ⇒ Uniform cost
- Suppose $g(n) = 1$, $h(n) = 0$ ⇒ Breadth first
- If non-admissible heuristic
  - $g(n) = 0$, $h(n) = 1/\text{depth}$ ⇒ depth first
- One code, many algorithms.

Behavior of A* search

Underestimation:

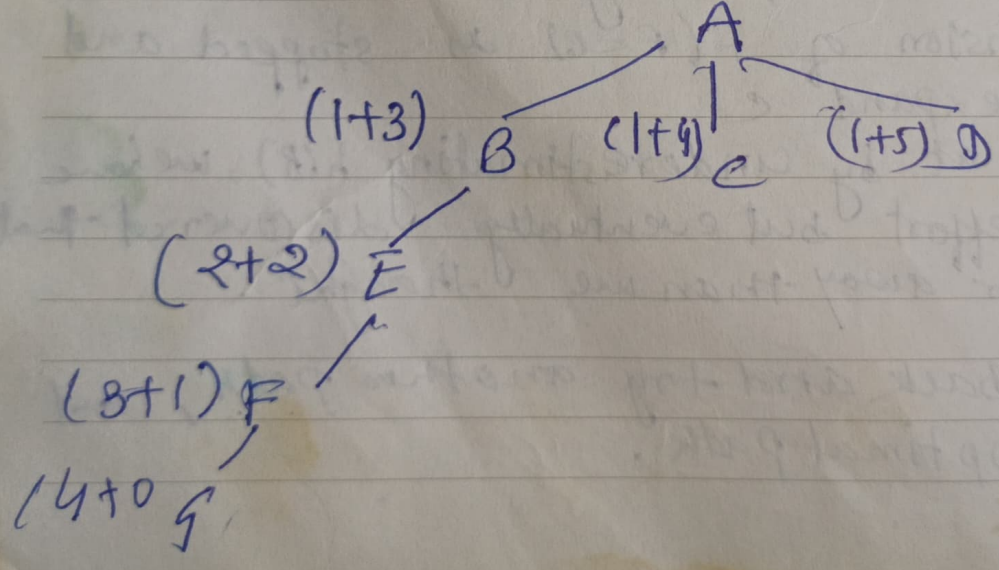
- If ~~one~~ we can guarantee that h never overestimates actual value from current to goal then A* algorithm is guaranteed to find an optimal path to a goal, if one exists.
- consider the following example.



- Assume the cost of all arcs is 1.
- We see that $f(E) = 5 = f(C)$
- Suppose we resolve in favor of E, the path currently we are expanding.
- Clearly expansion of f(f=6) is stopped and we will now expand C.
- Thus we see that by underestimating h(B), we have wasted some effort but eventually discovered that B was further away than we thought.
- Then we go back and try another path, and will found optimal path.

Overestimation :-

- Now consider another situation. We expand B to E, E to F and F to G for a solution path of length 4. But suppose that there is a direct path D to a solution giving a path of length 2.
- We will never find it because of overestimating h(D). We may find some other worse solution without ever expanding D.
- So by overestimating h, we can not be guaranteed to find the cheaper path solution.



Admissibility of A*

- A search algorithm is admissible, if for any graph, it always terminates in an optimal path from initial state to goal state, if path exists.
- If heuristic function h is underestimate of actual value from current state to goal state, then ~~the~~ it is called admissible function.
- So, we can say that A* always terminates with the optimal path in case h(x) is an admissible heuristic function.

Monotonocity:

- A heuristic function h is monotone if

1 ∀ states X_i and X_j such that X_j is successor of X_i
$h(X_i) - h(X_j) \leq cost(X_i, X_j)$ i.e. actual cost of going from X_i to X_j.

2 h(goal) = 0.