Page No.
Date

Till now,

State Space Search → Heuristic Search → optimization

↓ find Solutions faster.

↓ optimize the heuris value/ evaluation fⁿ

Let's study Solutions to find least cost. (This is critical in many domains)

⇒ Simulated annealing, ant colony optimization, Genetic algos doesn't guarantee optimality.

⇒ We want Solⁿ that guarantees optimality.

⇒ Here, we will have cost (weight to edge in Search Space)

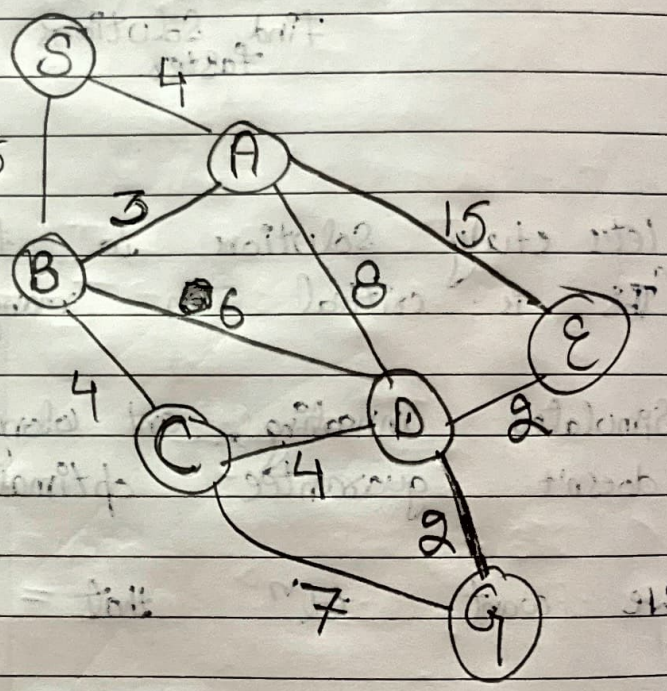
General Algo

Refine/Extend the least (estimated) cost partial Solution till such a Solⁿ is fully refined.

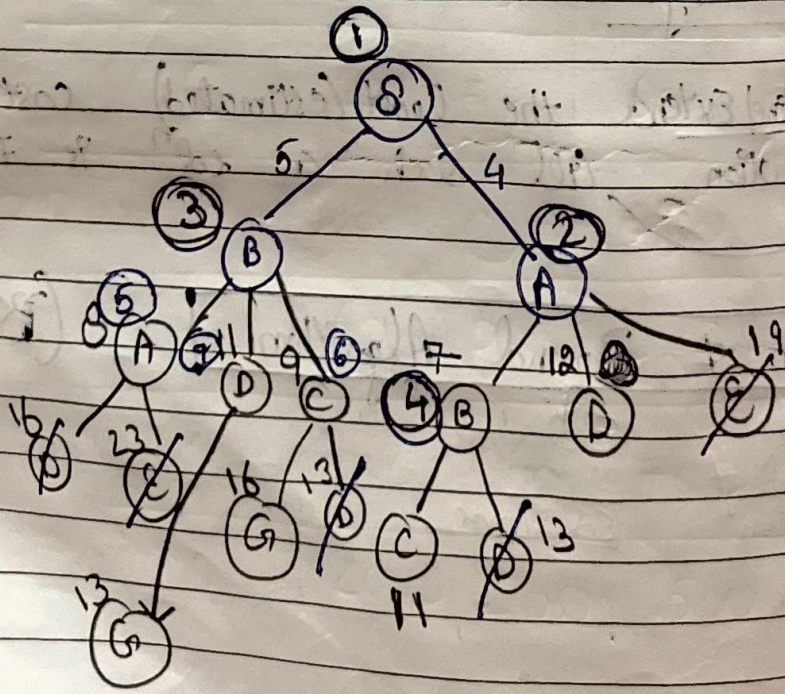
Branch & Bound Algorithm (B & B)

↓ Process of refinement/ extension

↓ Excluding some Solutions



Pick the lowest known partial sol^n cost and extend it



After (6), one Goal reached → Cost ⇒ 16
Eliminate all nodes with cost ≥ 16

Now, After (9) G1 ⇒ 13
eliminate all nodes with cost ≥ 13

Bound

This will keep on going till we explore all sol^n to Goal state
Final answer ⇒ least Goal State Sol^n.

⇒ ⇒ I will eliminate those nodes whose ~~actual~~ estimated cost is > some known sol^n

Here, we had actual costs, what if actual cost not given, we use some mechanism to find estimated cost.

Say estimated cost ⇒ $\hat{c}$
Actual cost ⇒ c^*

If $\hat{c} = c^*$ (Perfect estimate) Very. V.V rare

If $\hat{c} > c^*$ (overestimate)

If $\hat{c} < c^*$ (Underestimate)

⇒ Say, B/w 2 estimating f^n, one which underestimates the cost, another one which overestimate the cost, which one to use.

~~Overestimated~~ Underestimated one (∴ better ans)

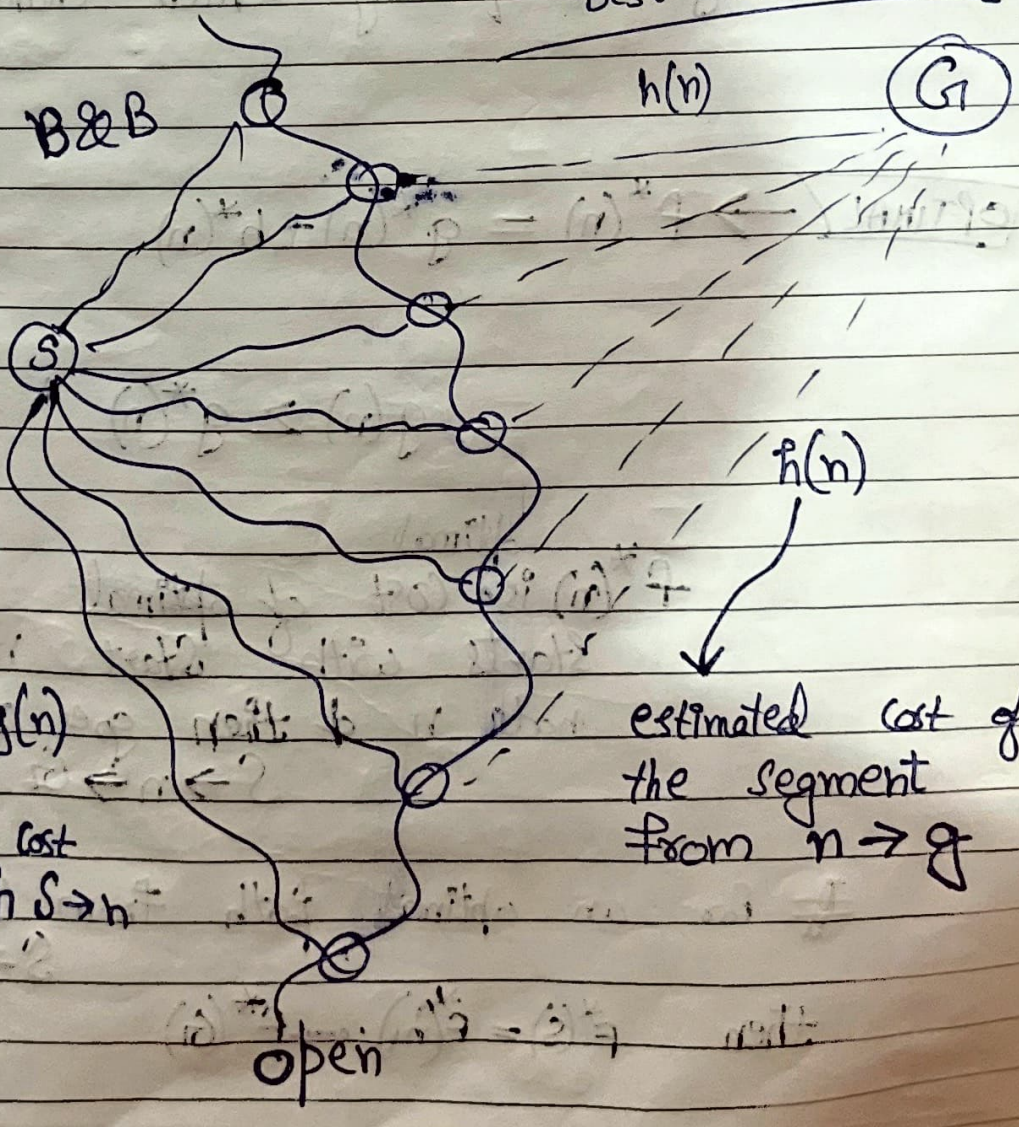
Page No.
Date

Through branch & bound, we will get optimal Sol^n always but it is not as fast as heuristics.

Graph Search

A* algorithm.

Best First Search



⇒ In B&B we try to stick as close to the source as possible (always picks node with lowest known cost)

⇒ BestFS always try to go as close to the goal as possible :: it uses only $h(n) = f^n$ which tries to be as close as possible to the goal

In A* we uses

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

We keep P.Queue sorted in values of f

g is a quantity which changes

OPTIMAL $\rightarrow f^*(n) = g^*(n) + h^*(n)$

$g(n) \geq g^*(n)$ Always

$f^*(n)$ is optimal cost of path that starts with Start node goes to node n & then goes to G

$S \rightarrow n \rightarrow G$

For an optimal path from $S \rightarrow G$

$S \rightarrow n_1 \rightarrow n_2 \cdots \rightarrow$

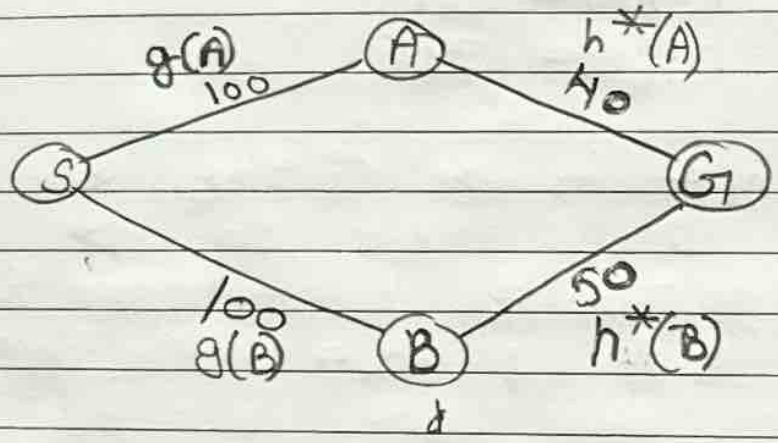
then $f^*(S) = f^*(n_1) \cdots f^*(G)$

Admissible means optimal sol? always

⇒ A^* is admissible if

$h(n) \leq h^*(n)$ → h_1 → Underestimates

$h(n) \geq h^*(n)$ → Overestimates → h_2



h_2

Let, $h_2(B) = 70 \quad > h^*(B)$

$h_2(A) = 80 \quad > h^*(A)$

$f_2(B) = 100 + 70 = 170$

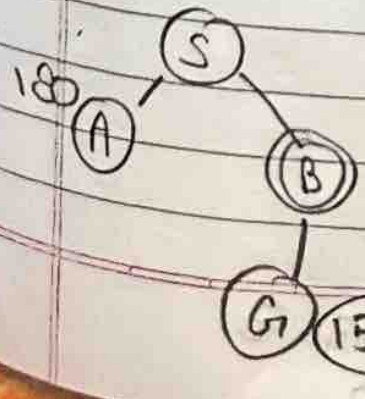
$f_2(A) = 100 + 80 = 180$

∴ Pick $f_2(B)$

But, $g_2(G) = g_2(B) + 50 = 150$

$h_2(G) = 0$

$f_2(G) = 150$



→ Pick this & we terminate

Now, h_1

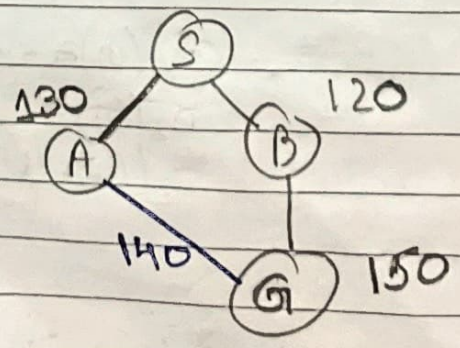
$h_1(n) \leq h_1^*(n)$

$h_1(B) = 80$
$h_1(A) = 30$

∴ h_1 & h_2 are similar both thinks B is closer to the goal only d/f $h_2 \Rightarrow$ overestimate
$h_1 \Rightarrow$ underestimates

$f_1(B) = 120$
$f_1(A) = 130$
Pick (B)

$g_1(G) = g_1(B) + 50 = 150$
~~$g_2(G) = g_1(A$~~
$h_2(G) = 0$
$f_1(G) = 150$



∴ Pick A picks G ←
Pick a & Terminate
∴ 140

Page No.
Date

∴ Underestimation → Admissible
of heuristics

⇒ Algorithm :-

OPEN ← S
parent (S) ← NIL
CLOSED ← NIL

best node n
↓
lowest f value
$f = g + h$

K → Cost of n

If OPEN ≠ NIL
Pick best node n & add
it to closed
If GoalTest(n) then reconstruct
Path(n)

else
Successors ← MoveGen(n)

For each m in Successors
Case 1 m ∉ OPEN & m ∉ CLOSED
Compute h(m)
Parent(m) ← n
$g(m) \leftarrow g(n) + K(n,m)$
$f(m) \leftarrow g(m) + h(m)$
Add m to open

Case 2 m ∈ OPEN
If $g(n) + K(n,m) < g(m)$
Parent(m) ← n
$g(m) \leftarrow g(n) + K(n,m)$
$f(m) \leftarrow g(m) + h(m)$

Case 3 m ∈ CLOSED
Do like in Case 2 if
better path found.
improve cost to

Proof of Admissibility of A*

Conditions for optimal path

Some +ve value epsilon

1. Finite branching factor.
2. Cost of each edge $k(m,n) > \varepsilon$
3. $h(n) \leq h^*(n)$ [Underestimating h]

If these 3 conditions are true, then A* always finds optimal path ie, it is admissible.

L1: Terminates for finite graphs

L2: At all times before termination there exists a node n' on OPEN which is on an optimal path.

One iteration
open → one node
taken out & put to closed

we will terminate if open empty or goal found

$S \to n_1 \to n_2 \cdots \to G$

Initially S on open
then n_1
then n_2
⋮
G

Further $f(n') \leq f^*(s)$

$f^*(s)$ → optimal cost of going from S to G

$$f(n') = g(n') + h(n')$$
$$= g^*(n') + h(n') \quad (\because \text{optimal path})$$
$$\leq g^*(n') + h^*(n') \quad (\because h(n) \leq h^*)$$
$$\leq f^*(n') = f^*(s)$$

where $f^*(n')$ is optimal cost of path that passes through n'.

(L3):- If there is a path to the goal the algo terminate (even for infinite graphs) with a path.

∵ At some point in P.Queue all nodes will be expensive then G_1.

(L4):- A* is admissible

Let A* terminate with a non optimal path such that

$$g(G_1) > f^*(S)$$

CANNOT ∵ it would pick n' (from L2)

n' is on open
& its value $\leq f^*(S)$
∴ it will be at head of PQ & not $g(G_1)$
∴ it will pick n'

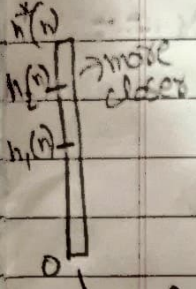
(5):- for every node n A* picks $f(n) \leq f^*(S)$

L6:-

If h_2 is more informed than h_1 for all nodes,

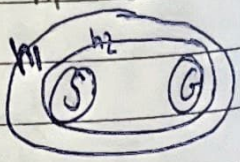
$$h_2(n) > h_1(n) \rightarrow *$$

Then any node seen by A_2^* is also seen by A_1^*



this means A_2^* will explore smaller part of search space & it has seen less.

→ P → If A_2^* sees a node then A_1^* also sees a node



P is proved for S

If A_2^* sees start node A_1^* also sees that

let, (P) be true for depth K

let, N be a node at depth (K+1) picked by A_2^*

let, A_1^* terminate without picking N

$$f_2(N) = g_2(N) + h_2(N) < f^*(S)$$

$$\therefore h_2(N) \leq f^*(S) - g_2(N) \rightarrow (1)$$

$$f_1(N) \geq f^*(S) \quad (\because \text{for } A_1^* \text{ N not picked})$$

$$f^*(S) \leq g_1(N) + h_1(N)$$
$$\leq g_2(N) + h_1(N)$$
$$h_1(N) \geq f^*(S) - g_2(N) \rightarrow (2)$$

$g_2(N) \geq g_1(N)$

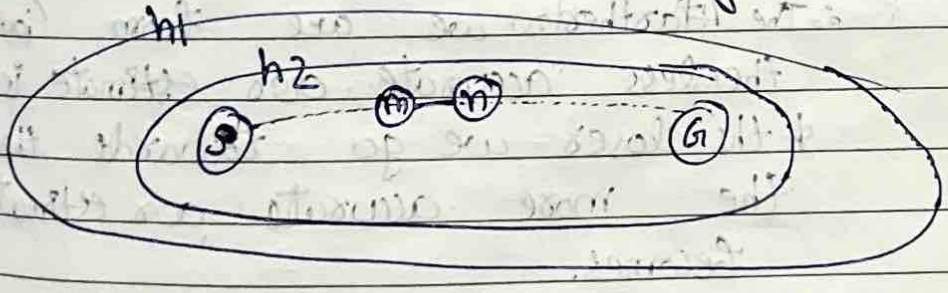
From ① & ②

$h_2(N) \leq h_1(N)$

Contradiction to $h_2(N) > h_1(N)$ [from *]

∴ If $h \uparrow$ the better it is for algo (less: seen elements)

⇒ A* Monotone Property



Some path from S to G

Say m & n are on this path

if it satisfies property of

$h(m) - h(n) \leq K(m,n)$

$h(m)$ → estimated cost from m to G

$h(n)$ → estimated cost from n to G

∴ $h(m) - h(n)$ sometimes is $K(m,n)$

sometimes in case of underestimation they are $< K(m,n)$

→ This condition is called monotone or Consistency condition.

If $h(m) - h(n) \leq K(m,n)$ is satisfied then whenever A* picks a node n it has already found an optimal path to n.

i.e., when we pick node n from OPEN list. At that point $g(n) = g^*(n)$

$$h(m) - h(n) \leq K(m,n)$$

$$\Rightarrow h(m) \leq K(m,n) + h(n)$$

Now, Add $g(m)$ on b/s

$$\Rightarrow h(m) + g(m) \leq K(m,n) + g(m) + h(n)$$

$$\Rightarrow g(m) + h(m) \leq g(n) + h(n)$$

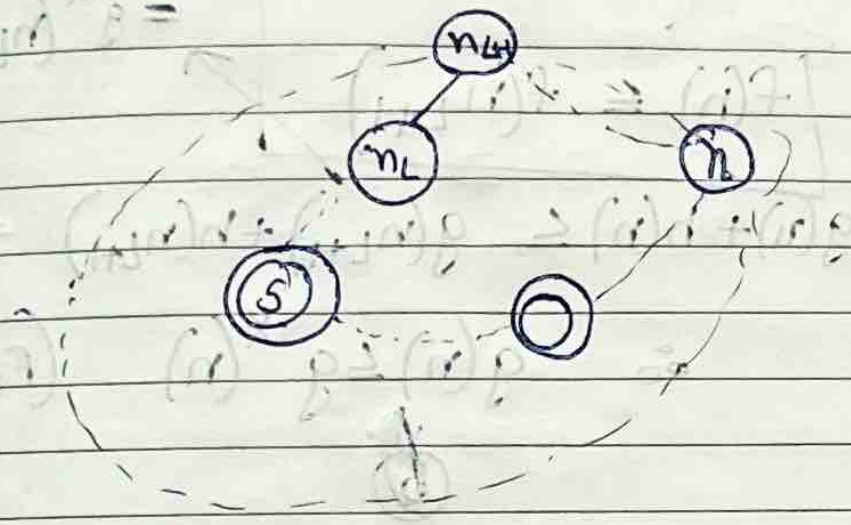
$$\Rightarrow f(m) \leq f(n)$$

∴ The farther we are from Goal, the less accurate our estimate is & the closer we go towards the Goal, the more accurate our estimate becomes.

∴ f values monotonically increases as we go towards the goal

Proof :- Say for a node n in open A* is about to pick n. We need to prove that if n is picked, then it must have found optimal path to n.

Assume that $g(n) > g^*(n)$. We will prove by contradiction



optimal path → $S \cdots n_L - n_{L+1} \cdots n$

Say A^* has found below path which is not optimal.

n_L → last node seen on optimal path

n_{L+1} → first node on optimal path to n which is not seen

$$f(n_L) \leq f(n_{L+1}) \cdots \leq f(n)$$

or $$g(n_{L+1}) + h(n_{L+1}) \leq g(n) + h(n)$$

Cause they are on optimal path

$$\boxed{g^*(n_{L+1}) + h(n_{L+1}) \leq g^*(n) + h(n)} \quad (1)$$

How ?

$f(n) \leq f(n_{L+1}) = g^*(n_{L+1})$

$g(n) + h(n) \leq g(n_{L+1}) + h(n_{L+1}) \rightarrow (2)$

$\therefore g(n) \leq g^*(n)$ (From (1) & (2))

Only possible if $g(n) = g^*(n)$

∵ optimal cost से < नहीं हो सकता।

Weighted A* algorithm

$f(n) = g(n) + K * h(n)$

If $K=1 \Rightarrow A^*$

If $K=0 \Rightarrow$ Branch & Bound

If $K \gg\gg$ (V. high) [Best First Search]

If $K>1$ then more emphasis to heuristic fⁿ (∵ we loss admissibility ∵ reach Goal as fast)

⇒ How to optimize space?

IDA* → Iterative Deeping A*

A* ⇒ Guarantee optimal solⁿ

ID → like DFID

In IDA* we have boundary.

⇒ Boundary is basically how far the start state thinks goal is.

In IDA*

bound = h(s)

Do DFS as long as f(n) ≤ boundary

increment bound

bound ← lowest unexplored f(n')

In DFID we increment bound by 1, here we increment bound to lowest f among all unexplored nodes (out of bound).

⇒ This guarantees optimal path ∵ $h(s) \leq h^*(s)$

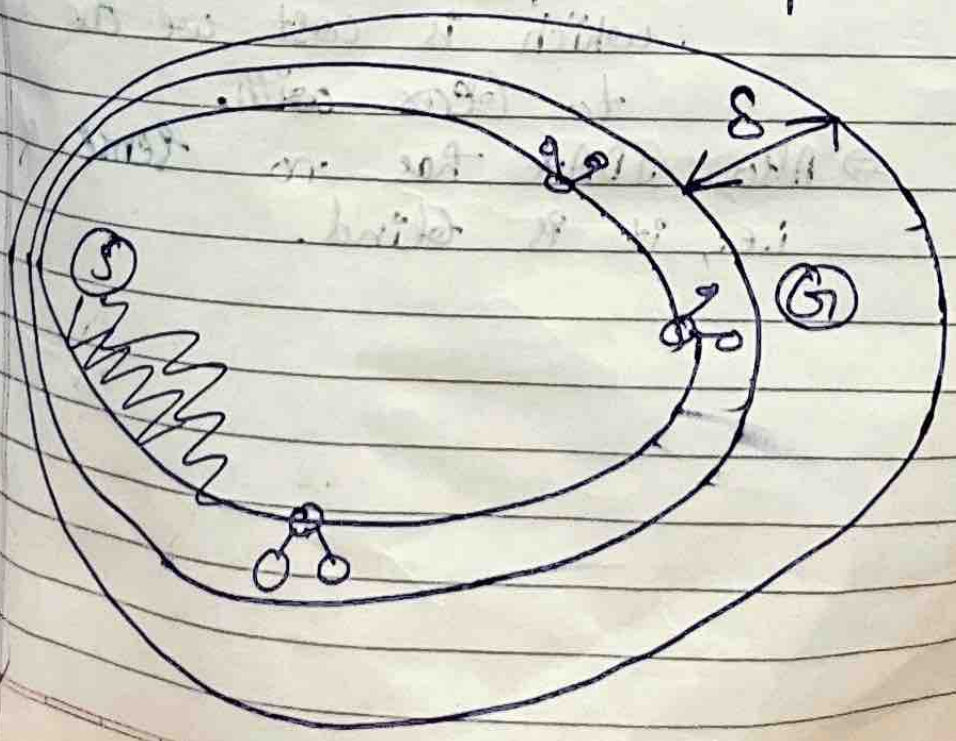
⇒ Save space ∵ DFS

⇒ T.C ↑ ∵ many iterations.

One way to tackle this is by incrementing bound by fixed δ.

∴ No of iterations ↓

In Worst case Sol^n = optimal + δ



This algo is uninformed (blind)
This algo uses h f(n) to determine boundary not to explore

Variations of A* that are space saving.

⇒ IDA* we have boundary, if sol^n not found boundary extended.

⇒ Space → Linear ∵ DFS

⇒ Optimal Sol^n → Yes, it finds optimal sol^n always.

⇒ T.C ↑

∵ we increase bound to lowest unexplored node ∴ we do DFS from start again. Many DFS repetetion held.

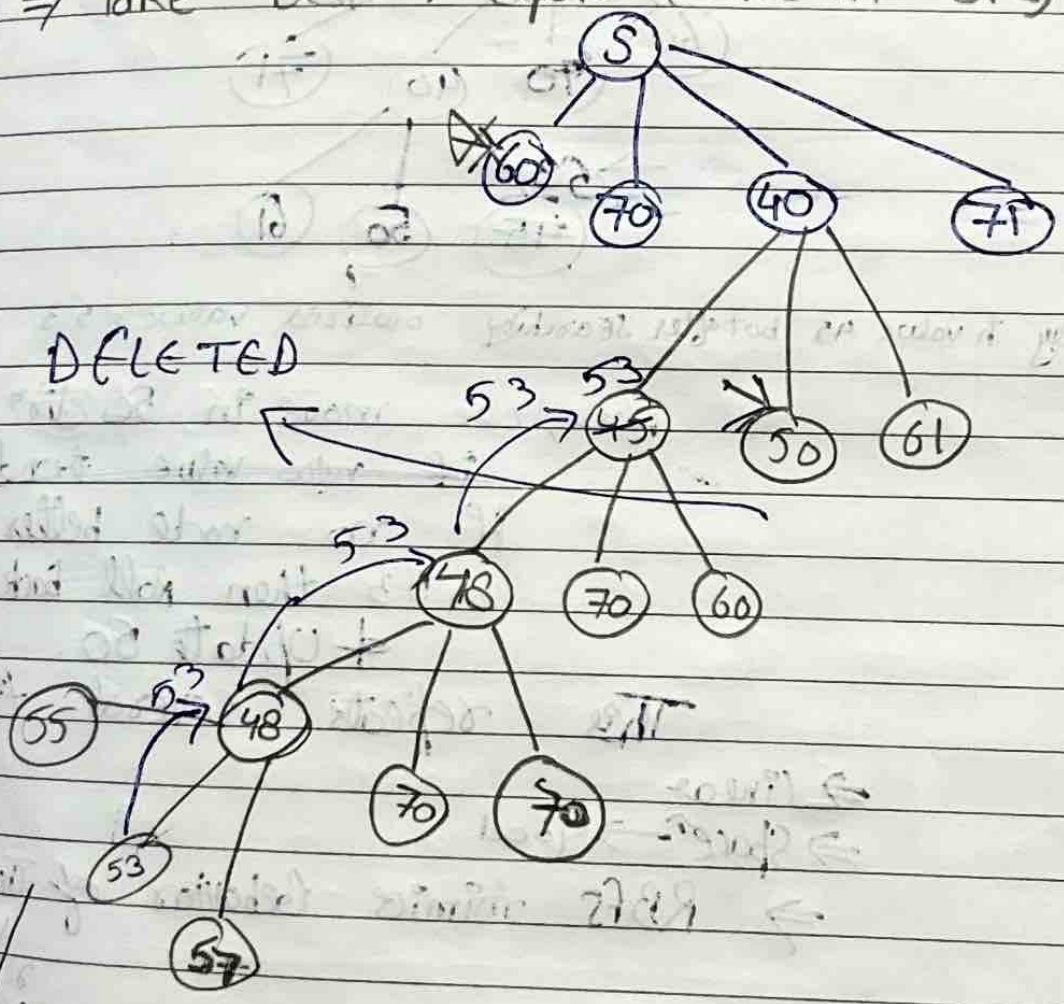
∵ Instead of increasing bound to lowest unexplored we increase it by δ which is cost we are ready to bear with.

⇒ Also, IDA* has no sense of dir^n i.e., it is blind.

Recursive Best First Search (RBFS)

⇒ Here, pointer to next best node

⇒ Take Best & expand (like in BFS)



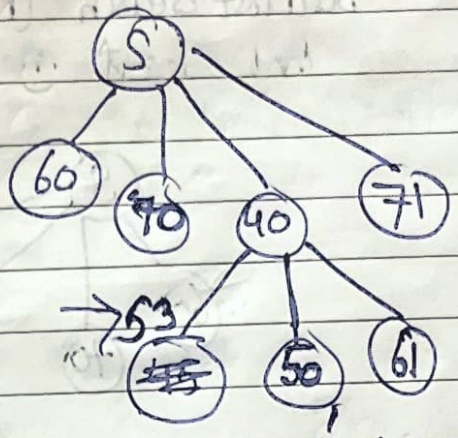
→ After this BFS would have picked Next Best node i.e 50 & expand it But, RBFS rolls back search to that it can reach next best. So it deletes nodes i.e., rolls back

⇒ Rule for Roll Back

BACKUP RULE

$f'(n) = f(n)$ of n in leaf or

$= \text{MIN } f'(\text{CHILDREN})$



Initially h value 45 but after searching realized value = 53

Now, we move in 50 dirn ~~if no value found~~ if no node better than 53 then Roll back to 53 & Update 50.

This repeats several times.

⇒ Linear
⇒ Space ↑ → Good
⇒ RBfs mimics behavior of Threshing. ↓ repeats several times but few nodes

T.C is ↑ then A* ∴ same space explored again & again.

SEQUENCE ALIGNMENT

↳ Allowed to insert Gaps (if alignment improves)

DNA Sequence alignment

ACGTCAGTCGTACG

ACG_CAGTCGTATCG

Two gaps included

In S.A we try to find some alignment that is optimal in some criteria.

⇒ To find which alignment is good or bad we have costs.

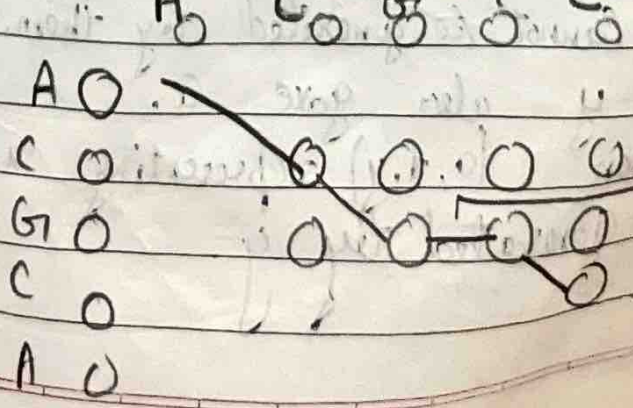
	Cost
Match	0
Mismatch	1
Gap	2

Based on these 3 operations, we want optimal cost.

From here this problem into Graph Search problem

Ways to solve this

⇒ Graph search



⇒ What if there is no CLOSE LIST?
Close Helps IN ① Avoid looping
② Find better paths.
③ Allows to reconstruct the path

② This can be done if monotone Condition is satisfied by h(n).
∴ Every n we take → part of soln. (∵ no close needed)
∴ We should worry about ① & ③

Now, we will assume that we are working with h(n) that satisfies the monotone criteria.

PRUNING The CLOSED LIST

① How to stop search from going backwards? / How to STOP "LEAKING BACK"? / How to STOP looping?

⇒ 1st Way → Store the nodes that should not be generated by MoveGen fn for that node with it in OPEN LIST only.

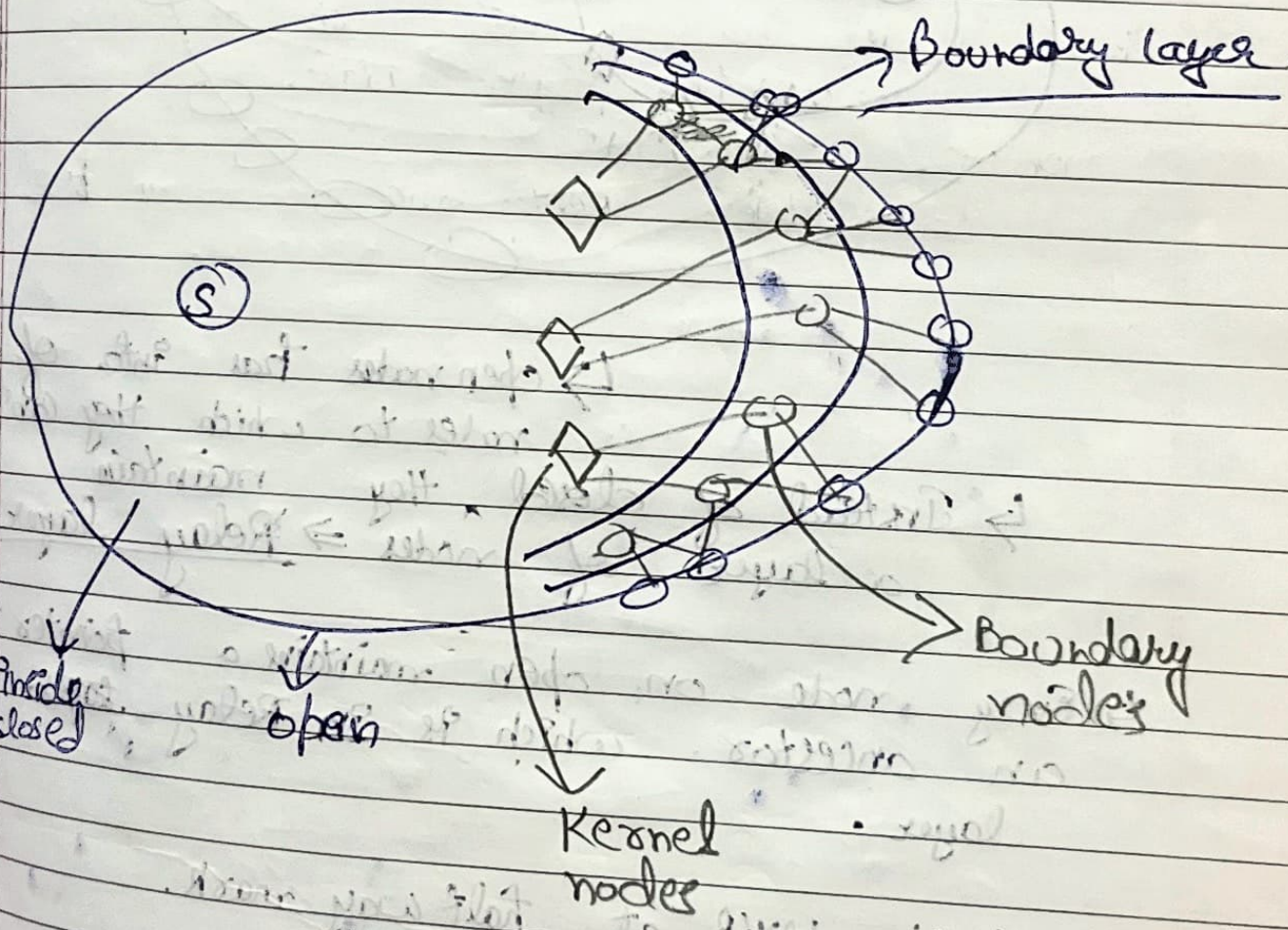
For eg → If MoveGen(x) gave a, b, c then put (a, x), (b, x), (c, x) in open representing x. Cannot be generated by them.
If y also gave a.
then (a, x, y) representing x, y cannot be generated by y.

2nd Way →

CLOSED

KERNEL — No child on OPEN

Boundary — atleast one child on open



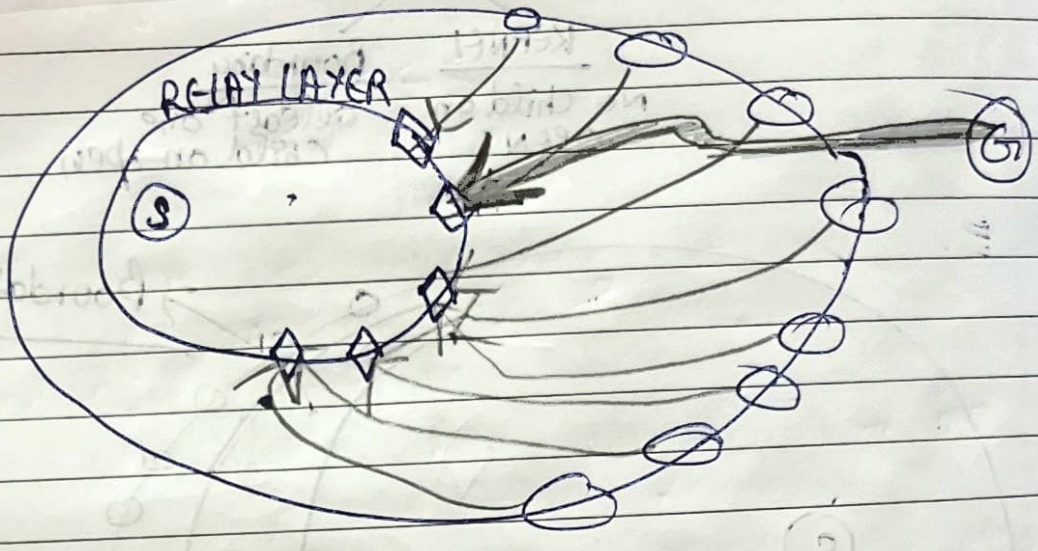
Idea is when we want to Gen. Child. of OPEN only look at boundary. Boundary server the old fn of CLOSED.

We don't want boundary nodes from OPEN.

② Reconstruct the path

Page No.
Date

In 1st Way → DCFS ⇒ Divide & Conquer Frontier Search



↳ open nodes has info of nodes to which they can't go

↳ Instead of closed they maintain a layer of nodes ⇒ Relay layer

↳ Every node on open maintains a pointer to an ancestor which is in Relay layer.

↳ RELAY LAYER at half way mark.

Half way mark when $g(n) \approx h(n)$

↳ Children of OPEN node will also point to some RELAY Node & so on.

↳ Goal will also point to some RELAY node.

↳ When we picked up (G) we know optimal cost to G but we don't know path. We do know one Relay node which is an ancestor of G that is it is on path to G.

Now, we want path $S \to G_1$

To reconstruct path we make 2 recursive calls to DCFS :-

One Call $\Rightarrow$ S to R
2nd Call $\Rightarrow$ R to G_1

This will give me 2 more nodes b/w [S & R] & [R & G_1]

Then make 4 recursive calls & so on
& keep doing this till the problem becomes edge case i.e next node is child of 1st one.

Space req. $\Rightarrow$ OPEN LIST, RELAY layer LIST

T.C $\Rightarrow$

If original problem with depth d had T.C $\to$ T(d)
then this one would have T.C $\Rightarrow$

$$T(d) + 2T\left(\frac{d}{2}\right) + 4T\left(\frac{d}{4}\right) + \cdots + xT(1)$$

$$= T(d) * \log_2(T(d))$$

d $\to$ exponential for min general

2nd Way →

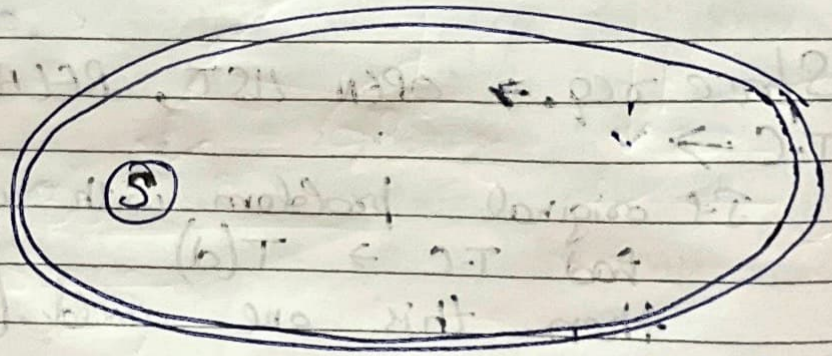
SMGS
Smart Memory Graph Search

Keep track of memory if running out of memory then prune the kernel nodes.

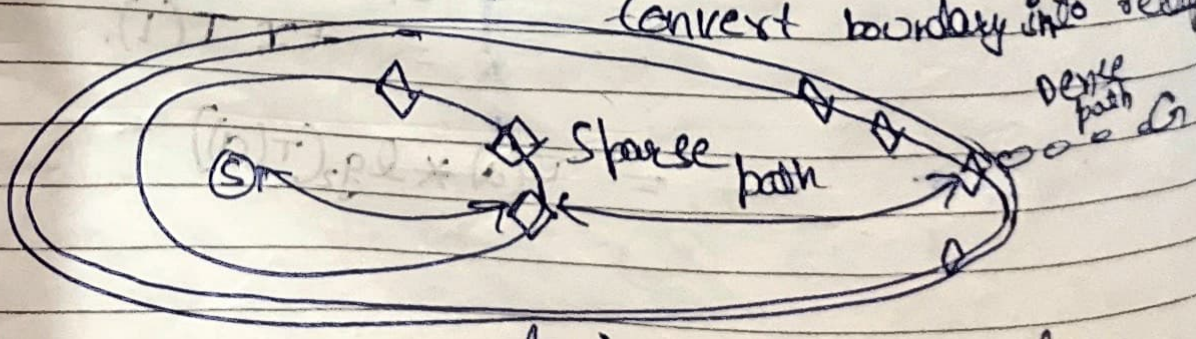
∴ We kept the boundary but pruned the Kernel.

& When we prune the Kernel convert Boundary into Relay.

& Search progresses from there.



Memory limit reached
Prune Kernel
Convert boundary into relay



Again memory reached
Same replat

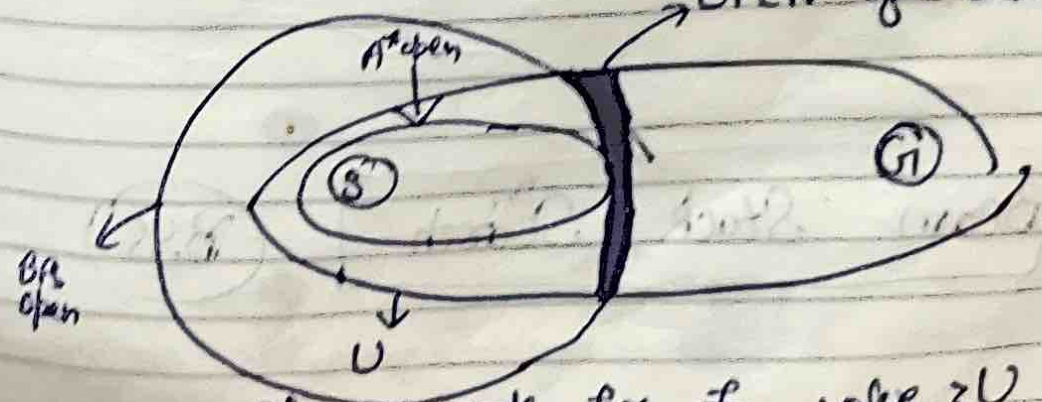
Here, many relay larger possible
maybe 0 or 1 or 2 ..... n

When Goal reached
we can move from Goal to
last relay then last relay to
other inner relays, hence path
reconstructed.
Here, Many recursive calls
to solve each of them.

PRUNING the OPEN LIST

① Breadth first Heuristic Search (BFHS)

U ; Compute U → Upper Bound
→ Maximum possible value that could be the cost of the solution.



If any node has f value > U, then
we won't expand it.

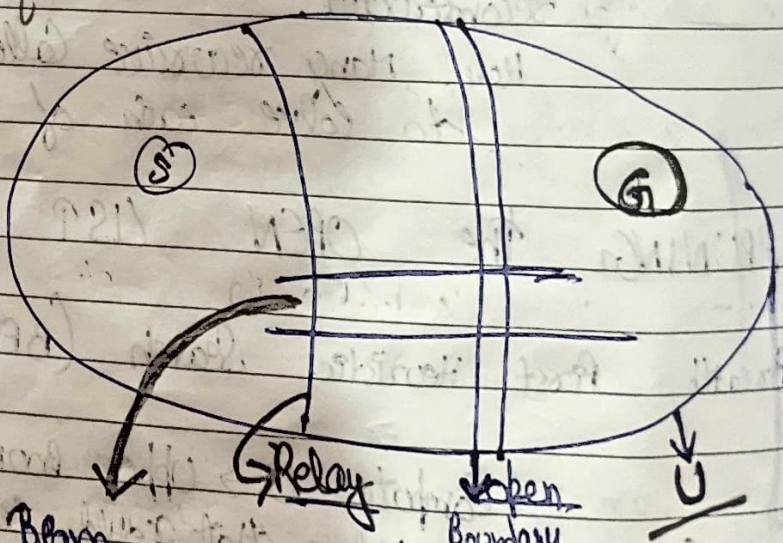
Page No.
Date

Variation → Beam Search
↓
Maintain OPEN LIST of constant width

Not Complete

↳ We can also convert this into DCBFHS

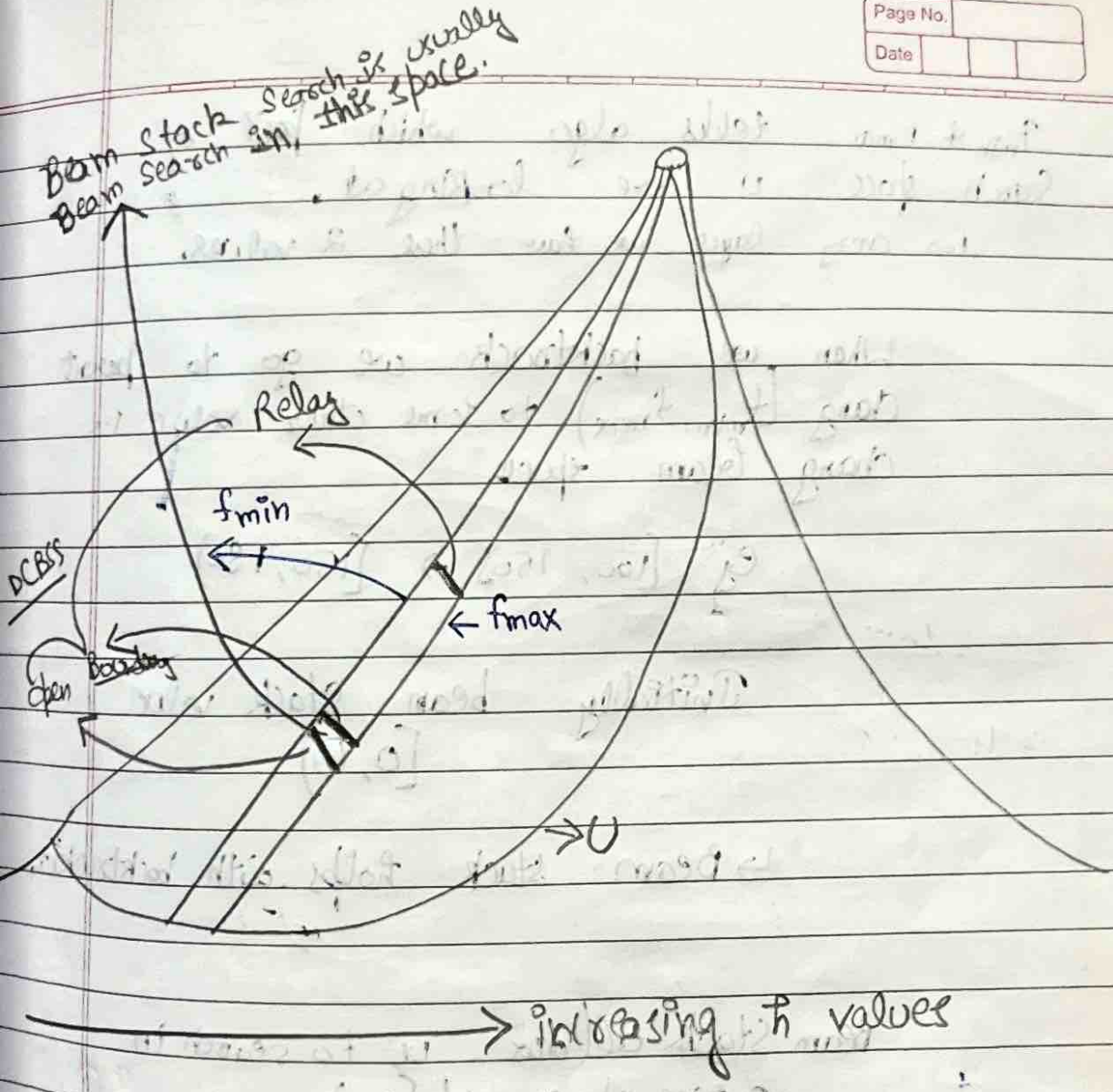
Divide & Conquer



Beam Search (Constant width)

② # Beam Stack Search (BSS)

Visualize nodes as tree



Beam Search is incomplete, we can make it complete by introducing backtracking

Backtrack after hitting boundary.

⇒ How to do Backtracking in this

Maintain another Data Structure → Beam Stack

At each layer we store 2 values

one is f_{min}

another is f_{max}

F_{min} & f_{max} tells algo which part of Search space u are looking at.

For every layer we have these 2 values.

When we backtrack we go to parent chang $[f_{min}, f_{max})$ to some other value i.e chang beam space

Eg $[100, 150] \rightarrow [150, 180)$

Initially beam stack value $[0, \infty)$

↳ Beam stack helps with backtracking

Beam stack allows us to search in entire space of $[0, \infty)$

Next ~~or~~ DCBSS

In ~~BSS~~ we can go to parent & regenerate ~~all the other beam~~ new set of children

In DCBSS we don't ~~have~~ have, we only have open bounds below layer

How to Backtrack?

- so go to start & use beam stack to generate till parent.
Beam stack will tell कि beam कहाँ से आया है.

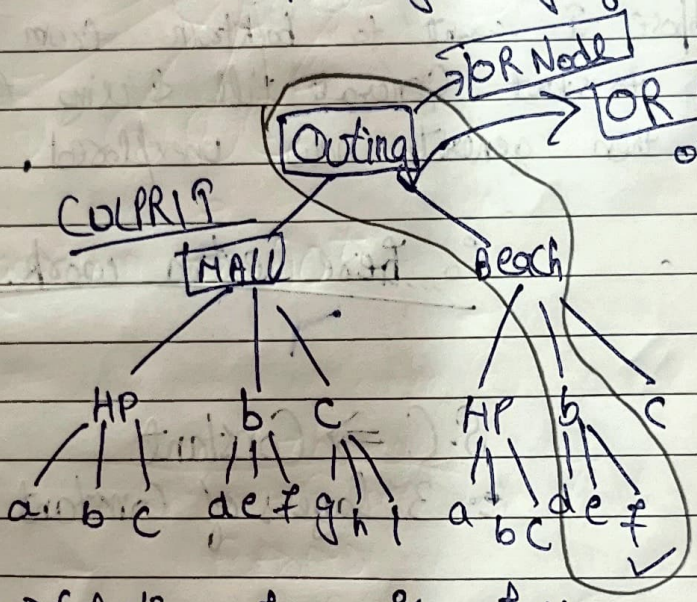
Suppose I want to backtrack from q to s
Go to start Generate till s using beam stack & then generate new unexplored.

So here extra work.

S.C ⇒ Constant
∴ 3. layers of constant width.

Problem Decomposition with Goal Trees

Say we have a problem of planning bday of friend



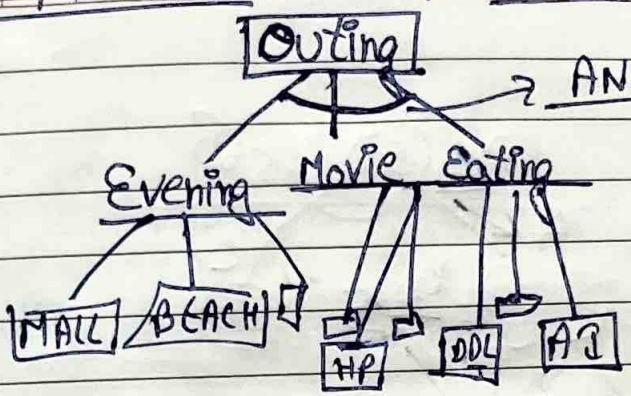
⇒ Solution here is path we only went to Outing - Beach - b - d, e, f after exploring complete left tree.

Problem was with MALL we should have backtracked from there only.

⇒ How to address this difficulty?

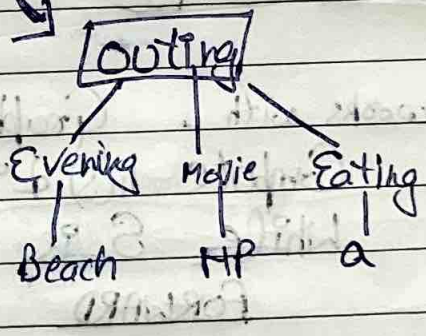
In CSP
1. Dependency Directed Backtracking
2. AND/OR Graphs / Trees OR Goal

AND/OR tree → AND Node



→ AND Arc (all)

⇒ Solution here is Subtree →

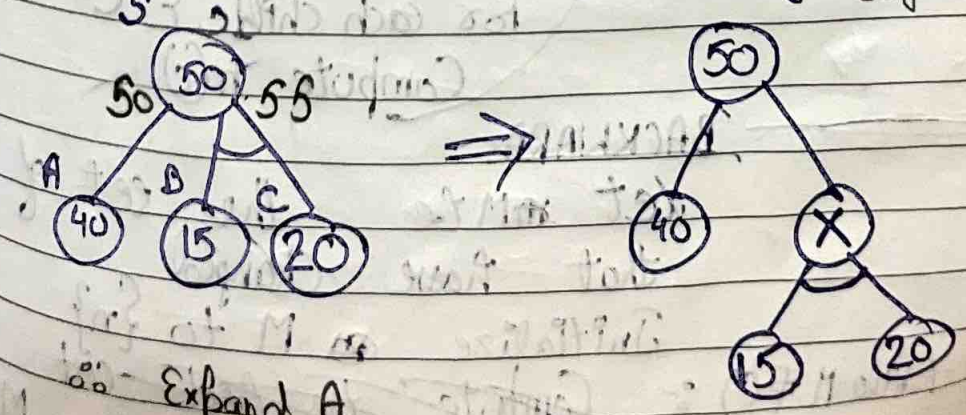


⇒ Imp. thing → We broke problem into 3 small problems & solved them independently of each other.

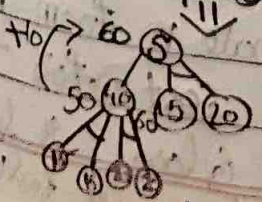
Algo → break problem into simpler problems.

h(n) → estimation of solving node n

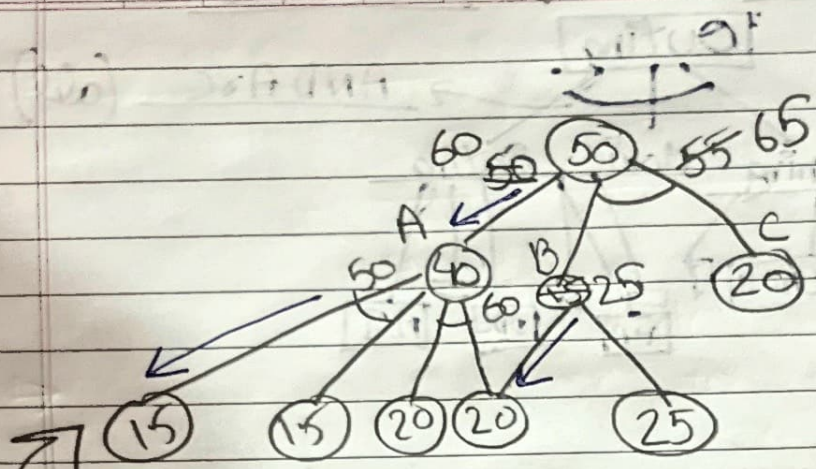
EdgeCost = 10 (for all edges)



∴ Expand A



Leaf → Solved



Algorithm → AO*

↳ AO* works with a Graph G initialized to (S)

Compute h(S) & initialize

While, S is not solved and h(S) < FUTILITY

FORWARD

Follow the markers to a set of nodes N.

Pick some n ∈ N

Generate children C of n

if C is none

h(n) ← FUTILITY

Remove loops

for each child c ∈ C

Compute h(c)

↳ label solve is not present.

↳ Futility = Some value above which we don't want any h(n)

↳ marker points to best node in that area.

make terminal node leaf

If C is primitive label c as solved

BACKWARD

Let M be the set of nodes that have changed

Initialize M to {n}

While M ≠ (') :-

Compute the best cost

Pick lowest node n from M

Compute best cost for n

If bestSol" below it labelled Solved, label n as solved.

We propagate back to parent & revise cost based on children produced in forward also change markers

Mark best hyperedge

Page No.
Date

If node n has changed then
add all parents of n to M

As long as $h(n) \leq h^*(n)$ we get optimal cost