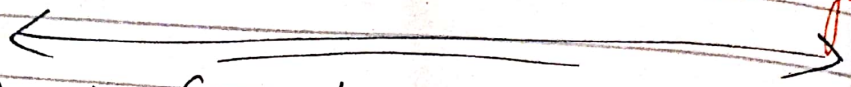
Note:- (why we square things?)
→ absolute error captures the error value but not the variation in the error vs. squared error that captures the variation in the error.

S-I :- Q1: chapter 1, Q2: chapter 2, Q3: ~~Soft~~ learning Based. ajuster



ANN :- (Artificial Neural Network) :-

1. Underfitting & Overfitting
2. ANN architecture
3. Back propagation Algorithm

Training data :- The data containing examples to learn the parameters of a classifier.
e.g. OR data to learn weight values

Test data :- The data containing previously unseen examples to test a classifier's performance.

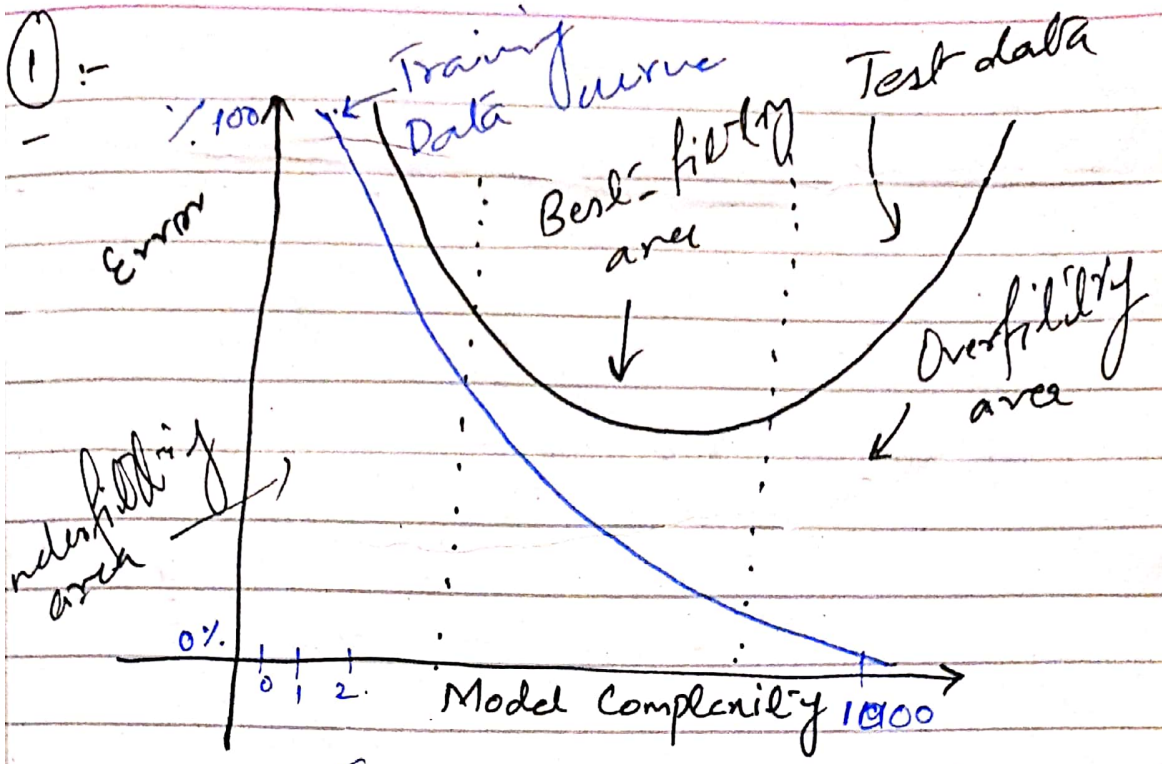
Training :-

	x_1	x_2	td
E_1	0	0	0
E_2	0	1	1

Test :-

	x_1	x_2	td
E_1	1	0	?
E_2			

(1) :-



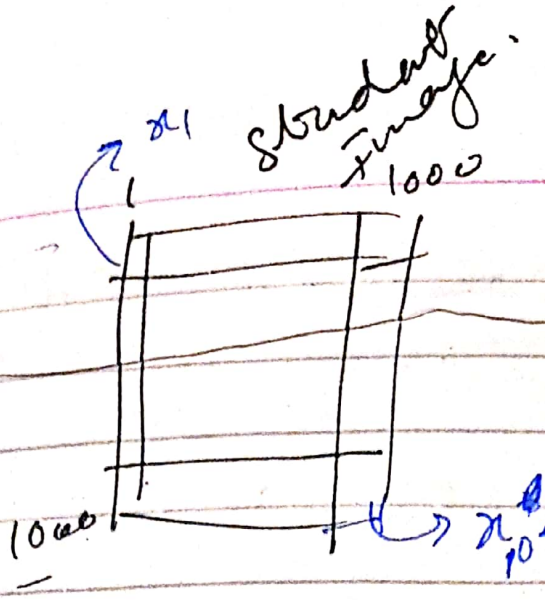
↳ No. of perceptrons ↑
No. of weights (features) ↑

Underfitting :- (Need to increase complexity)
High error on the training & the test data.

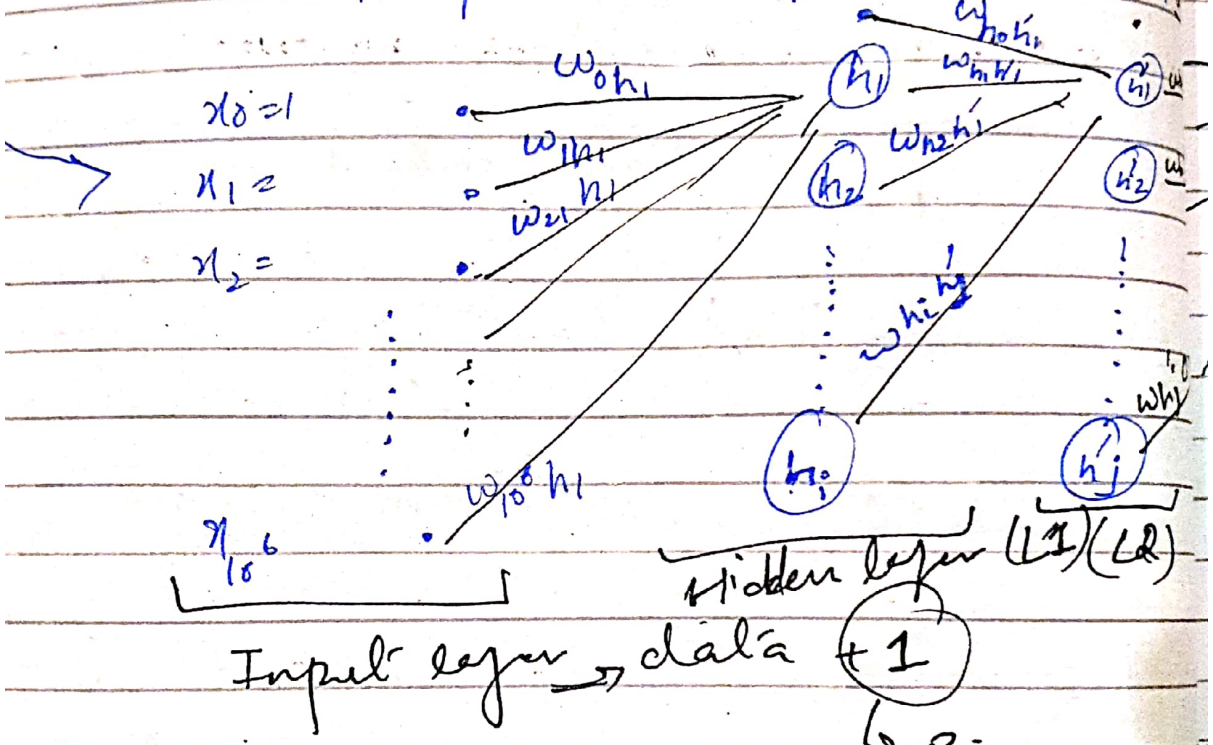
Overfitting :- (Need to reduce complexity)
Low error on the training data & High error on the test data.
Remedy :- or increase the training data.

Best-fitting :- Acceptable error on both the training & test data.

2):-



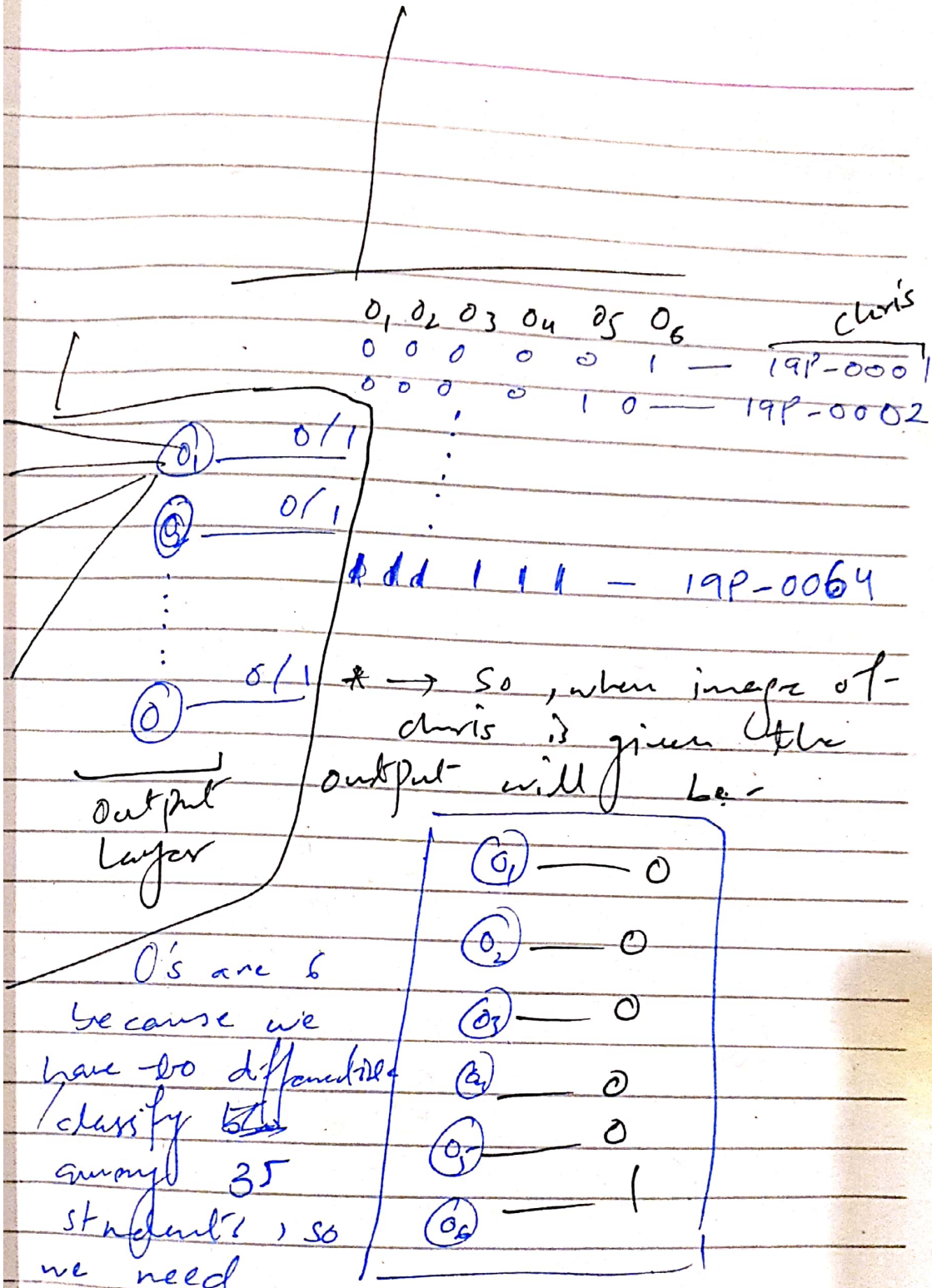
total pixels = 1,000,000
= 10^6 pixels.



→ Why Bias?

↳ In case of a suppressed signal, we need to elevate the resultant.

* → Almost all the time only 2 hidden layers are sufficient.



Chris

* → So, when image of Chris is given the output will be -

0's are 6 because we have to differentiate/classify 35 students, so we need 6 bits (64 bits) using more bits will result in overfitting.

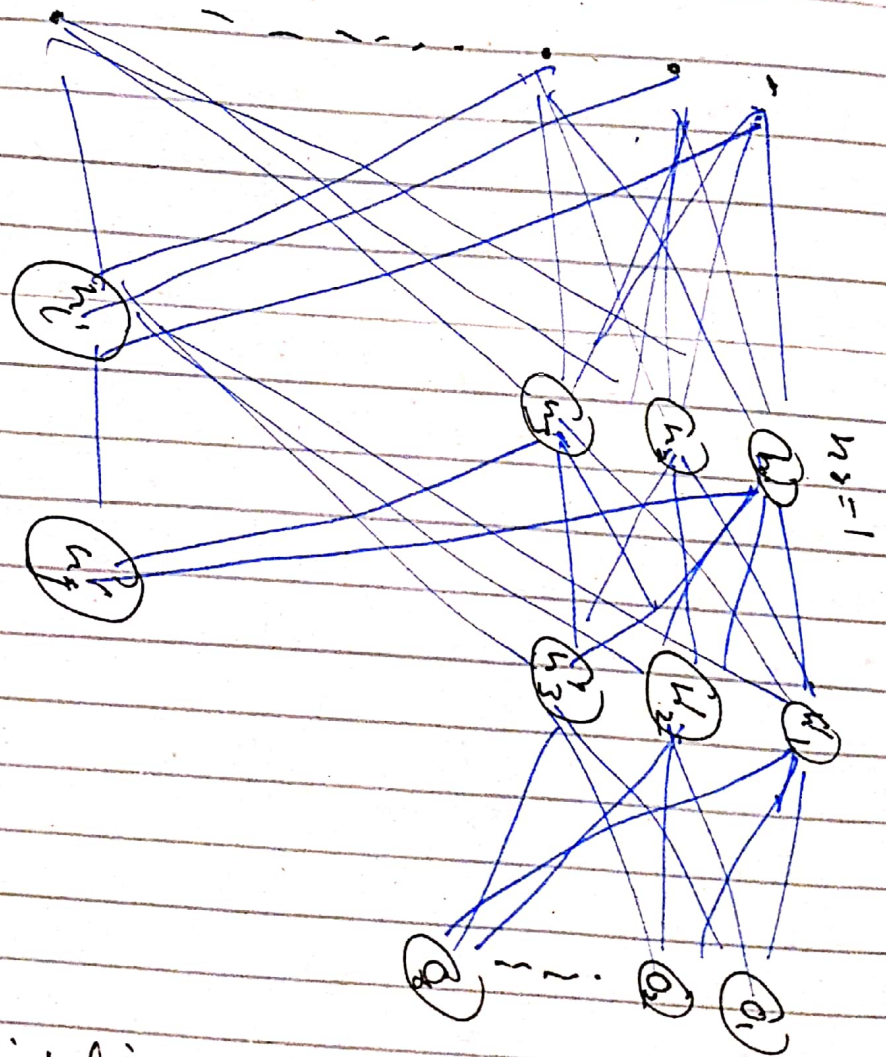
→ Perceptron count of Hidden Layer & Layer counts are dependent on the Over, underfitting graph. So start with low & keep observing.

ANN

Complete architecture:-

The same algorithm but we use back propagation, by propagating from output to see how each hidden perceptron affects/contributes

if $w_{oh_1} = -\eta \frac{\partial E}{\partial w_{oh_1}}$



validation dataset is used to fine-tune our hyperparameters.

All that are representative of our purpose.

Artificial Neural Network:-

Back propagation Algorithm:-

1. Initialize all weights randomly i.e w_{ij}
2. Until convergence, Do
3. Input it to network & compute network outputs.
4. For each output unit k.

$$\delta_k \leftarrow o_k (1 - o_k)(t_k - o_k)$$

5. For each hidden unit h.

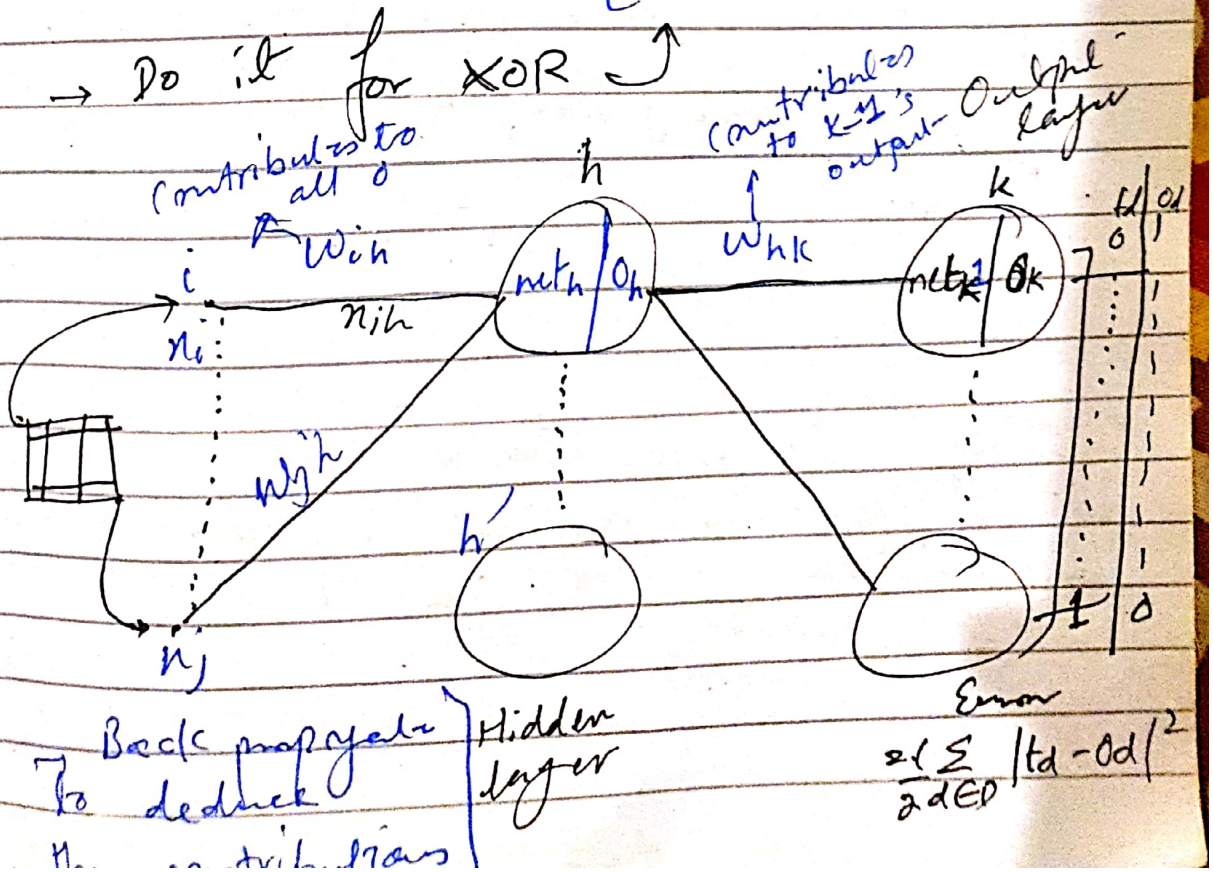
$$\delta_h \leftarrow o_h (1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

6. Update each network weight w_{ij}

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

→ Do it for XOR



Weight update for output layer:-

$$\Delta w_{hk} = -\eta \cdot \frac{\partial E}{\partial w_{hk}} = -\eta \left(\frac{-\partial E}{\partial net_k}\right) \cdot \frac{\partial net_k}{\partial w_{hk}}$$

$\left(\frac{-\partial E}{\partial net_k}\right) = \delta_k = o_k$

$$\delta_k = o_k = \frac{\partial E}{\partial net_k} = \frac{\partial}{\partial \cdot net_k} \cdot \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \sum_{d \in D} (t_d - o_d) \cdot \left(0 - \frac{\partial}{\partial net_k} o_d\right)$$

$$= \sum_{d \in D} (t_d - o_d) \cdot \left(0 - o_d(1 - o_d)\right)$$

$$\left(-\frac{\partial E}{\partial net_k}\right) = \sum_{d \in D} (t_d - o_d)\, o_d (1 - o_d)$$

$$\boxed{o_k = \sum_{d \in D} (t_d - o_d)\, o_d (1 - o_d)}$$

for all training examples

$$\boxed{\delta_k \leftarrow o_k (1 - o_k)(t_k - o_k)}$$

* This is the derivative of sigmoid, so it can be replaced by other functions' derivatives such as relu etc.

→ for each O_k in the image example once you get $O_1 \to O_6$ then you can back propagate to find out contribution of the error in hidden layer perceptrons.

→ See back that:

$$net'_k = w_{hk} \cdot O_h + \cdots + w_{h'k} \cdot O_{h'}$$

$$\boxed{\frac{\partial net_k}{\partial w_{hk}} = O_h} \quad //$$

$$\Delta w_{hk} = \eta \left(\underbrace{\frac{-\partial E}{\partial net'_k}}_{\downarrow \; \delta_k} \right) \cdot \underbrace{\frac{\partial net_k}{\partial w_{hk}}}_{\downarrow \; O_h}$$

$$\uparrow$$

$$O_k (1 - O_k)(t_k - O_k)$$

Now for :- (weight update for hidden layer)

$$\Delta w_{ih} = -\eta \cdot \frac{\partial E}{\partial w_{ih}} = \eta \left(\underbrace{\frac{-\partial E}{\partial net_h}}_{\delta_k} \right) \cdot \frac{\partial net_h}{\partial w_{ih}} \quad \downarrow \; x_{ih}$$

$$\frac{\partial E}{\partial net_h} = \frac{\partial}{\partial net'_h} \times \frac{1}{2} \sum_{d \in D} (t_h^{(d)} - O_h^{(d)})^2$$

$$= \frac{\partial E}{\partial O_h} \cdot \frac{O_h}{net_h} = \frac{\partial E}{\partial net_k} \cdot \sum_{k \in Out} \frac{\partial net_k}{net_h}$$

$$\frac{\partial E}{\partial net_h} = \frac{\partial E}{\partial O_k} \cdot \sum_{out \in K} \cdot \frac{\partial net_k}{O_h} \cdot \frac{O_h}{net_k}$$

$$\frac{\partial E}{\partial net_h} = \sum_{k \in Out} \frac{\partial E}{\partial net_k} \cdot \sum_{k \in Out} \frac{\partial net_k}{net_h}$$

$$= \sum_{k \in Out} (-\delta_k) \cdot \sum_{k \in Out} \cdot \frac{\partial net_k}{\partial O_h} \cdot \frac{\partial O_h}{net_h}$$

$$= (-\delta_k) \cdot (w_{hk}) \cdot O_h (1 - O_h)$$

$$\boxed{\delta_h = O_h (1 - O_h) \cdot \sum_{k \in Out} \delta_k \cdot w_{hk}}$$

$$\Delta w_{in} = \eta \cdot \delta_h \cdot x_{in}$$

Book chapter 4

Output → Hidden → ..... → hidden

End of Artificial neural network.

Weight Update for Hidden Layer..

$$\text{Error} = \frac{1}{2} \sum_{d \in D} \sum_{j=k}^{k'} \left(t_j^{(d)} - o_j^{(d)}\right)^2$$

$$\delta_k = -\frac{\partial E}{\partial net_k} \quad ; \quad \delta_h = -\frac{\partial E}{\partial net_h}$$

$$\Delta w_{ih} = -\eta \cdot \frac{\partial E}{\partial w_{ih}} = \eta \cdot \left(-\frac{\partial E}{\partial net_h}\right) \cdot \frac{\partial net_h}{\partial w_{ih}}$$

δ_h

$$\frac{\partial E}{\partial net_h} = \frac{\partial}{\partial net_h}\left(\frac{1}{2} \sum_{d \in D} \sum_{j=k}^{k'} \left(t_j^{(d)} - o_j^{(d)}\right)^2\right)$$

$$\frac{\partial E}{\partial net_h} = \sum_{k \in outputs} \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_h}$$

$$= \sum_{k \in outputs} \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_h} \cdot \frac{\partial o_h}{\partial net_h}$$

$$= \sum_{k \in outputs} (-\delta_k) \cdot w_{hk} \cdot o_h(1 - o_h)$$

$net_k = o_h w_{hk} + \dots + o_h w_{h'k}$

$\frac{\partial net_k}{\partial o_h} = w_{hk}$

$$\Delta w_{ih} = +\eta \cdot \frac{-\partial E}{\partial net_h} \cdot \frac{\partial net_h}{\partial w_{ih}}$$

x_{ih}

$$\delta_h = \sum_{k \in outputs} \delta_k \cdot w_{kh} \cdot o_h(1 - o_h)$$

Fractional effect

$$\boxed{\Delta w_{ih} = \eta \cdot \delta_h \cdot x_{ih}}$$

Artificial Intelligence:-

=

Hypothesis Spaces:-

* Hypothesis:- A hypothesis is a fn that approximates the true fn (ground truth) by minimizing error.

→ If hypothesis fn = true fn for all data, then error = 0%.

e.g OR //

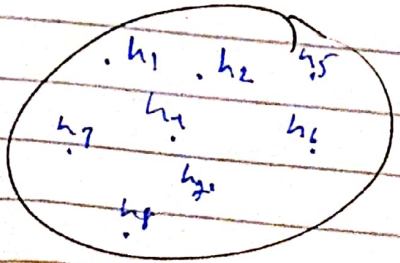
x_1	x_2	t	h_1	h_2
0	0	0	1	0
0	1	1	0	1
1	0	1	1	1
1	1	1	1	1
			50%	0%

→ Every eq to predict the target data is a hypothesis.

Hypothesis Space:-

=

is a huge space containing different hypothesis. (N number)



*

→ Making a hypothesis is hard from the entire hypothesis space that is optimal & minimizes the error.

e.g.:- for a binary one feature,

x_1	h_1	h_2	h_3	h_4
0	0	0	1	1
1	0	1	0	1

→ If target variable is binary, 4 hypothesis are possible in the hypothesis space.

for 2 features:-

x_1	x_2	h_1	h_2	h_3	h_{16}
0	0	0	0	0	1
0	1	0	0	0	1
1	0	0	0	1	1
1	1	0	1	0	1

General:-

x_1	x_2	x_3	h_1 h_2 .... h_{28}
0			
0			
0			
⋮			

8

* Hypothesis space size for binary features:-

(# of features) → size of truth table

Hypothesis space size = $2^{2^1} = 2^2 = 4$

$= 2^{2^2} = 2^4 = 16$

$= 2^{2^3} = 2^8 = 256$ possible hypothesis

Generic (Non-binary):-

* (# of classes/labels) ^ (Types of features) ^ (# of features)

eg: 10 possible values of labels
" " " features
5 number of features

$\Rightarrow (10)^{(10)^{(5)}}$

→ Gradient descent also searches in this space by updating weights.

* Okhem's Razor Principal:-

=

o Actual
x predicted

$h_1 = w_0 x_0 + w_1 x_1$ E = 10%

$h_2 = w_0 x_0^2 + w_1 x_1$ E = 2%

$h_3 = w_0 x_0^3 + \dots$ E = 2%

If 20% error is acceptable & all the hypothesis above fit the data-points equally acceptably then choose the simplest.

Razor

Ockham's principle--

→ If multiple hypothesis give acceptable error then choose the simplest one. (Wire fox in our example).

(Table)

* Instance Based Classifiers:- (Lookup algo)

① K-Nearest Neighbor (Non-parametric)

Training:

	x_1	x_2	x_3	label
E_1	0	0	0	A
E_2	0	1	0	B
E_3	0	0	1	B
E_4	1	1	0	C

Test:

x_1	x_2	x_3	label
0	1	0	?

* → lookup table will search the table for the test input so it will assign B label. But it will fail in case no match is found. So, we go for KNN.

KNN :-

1. Training data
2. The value of K (usually odd, to avoid ties).
3. Distance metric (We will use Hamming distance).

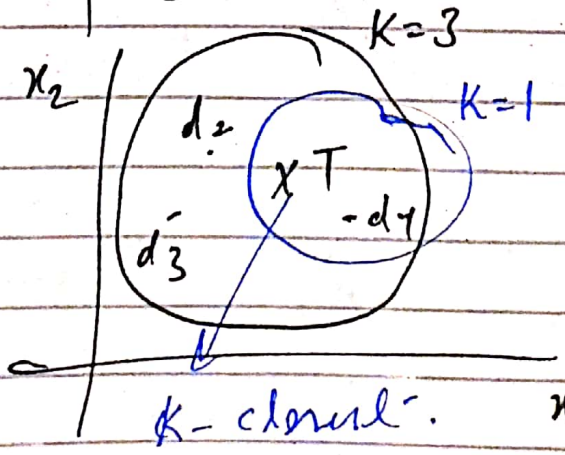
e.g.:- K =

E_1 :-	0	0	0
T	1	1	1

$d_1 = 3 = 1 + 1 + 1$

distance		x_1	x_2	x_3	label
d_1	3				A
d_2	2				B
d_3	2				B
d_4	1				C

Previous table



(Tie in x_3 is not considered)

		x_1	x_2	x_3	label
Eg.	T	1	1	1	?

For K=1; label = C

For K=3; label = B

Think & group rich

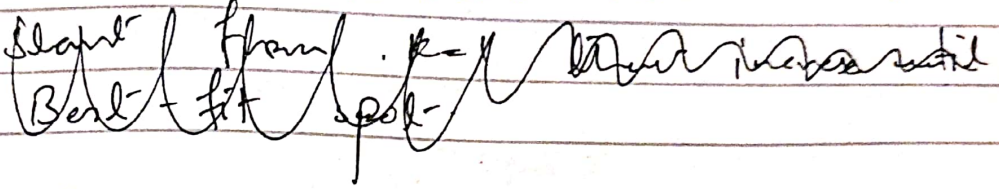
Techniques to break tie:-

① Decrease K.

② Take avg distance of the tied points within & then choose the smallest- avg label.

Full data → Training data

→ Validation data



③ Select a class randomly (if no other choice).

* Distance Metric:-

① Hamming Distance: Binary attributes

② Euclidean Distance: if attribute units are the same.

eg Training → Q

Test → E

$$d(Q,E) = \sqrt{\sum_{i=1}^{\text{attributes}} (Q_i - E_i)^2}$$

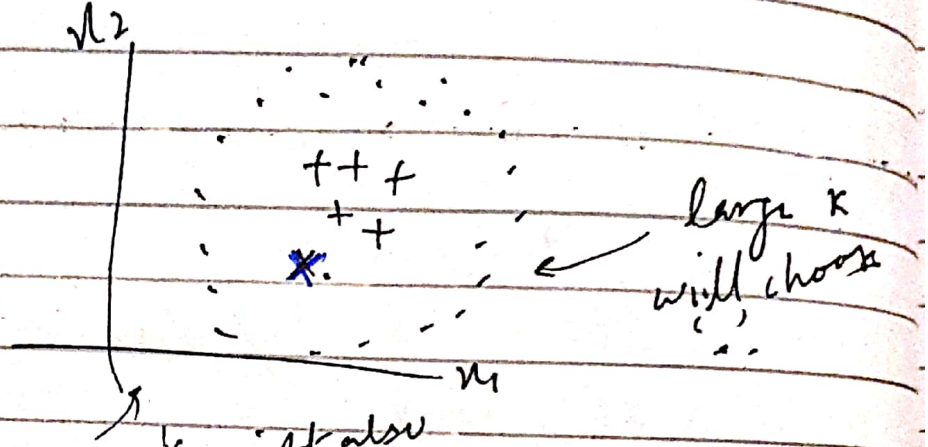
③ Manhattan Distance: attribute units are real numbers & not the same.

$$d(Q,E) = \sum_{i=1}^{\text{attributes}} |Q_i - E_i|$$

Choosing value of K:-

① If K is too small, the classifier is sensitive to noise.

② If K is too large the classifier will classify into the majority class.



height	distance
x_1	x_2
1m	500m
1.5m	800m
1.4m	700m
1.6m	900m

In this case x_2 has a much higher value. & so it will have a higher effect. Hence, we need to normalize it.

Scaling / Normalization Issues:-

P(2, 3) ——— ?

x_1	x_2	label
-4	3	A ← $\sqrt{36}$
6	3	B ← $\sqrt{16}$

→ Euclidean Distance from both

→ classified into the label B.

→ also change dimensions in some cases e.g. convert ~~meters~~ km into ~~km~~ meters. or meters into cm.

x_1	x_2	label
8	3	A ← $\sqrt{162}$
-12	3	B ← $\sqrt{142}$

→ Classified into the label A.

Reasons: So that one attribute doesn't dominate some other attr.

Ways:-

① Division by the maximum.

$$x_i = [x_i - \min(x_i)] / \max(x_i)$$

(2) Z-Score method.

$$x_i = \left| \frac{x_i - \mu_i}{\sigma} \right|$$

mean.

Problems with Euclidean Measure:- (KNN)

(1) → KNN works well in low dimen. spaces but for higher dimensions it suffers from curse of dimensionality.

(2) Higher dimensions increase the distances.

(3) Higher dimensions make datapoints outliers close to eachother. Eg Thus decreases prediction accuracy.

* KNN also * are lazy learners. unlike eager learners such as Neural networks.

Chapter # 03:-

Problem Solving by Searching:-

★ Searching Agents:-

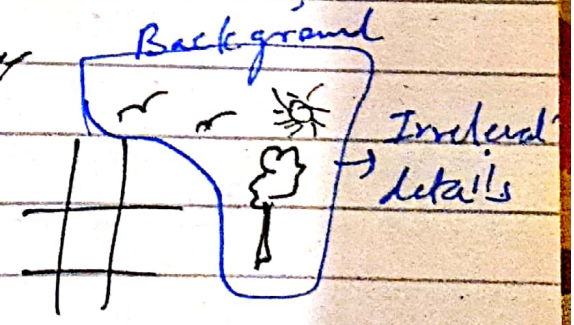
① They have a goal to achieve.
② The effects of actions is known. i-e, the agent knows how the world evolves.
③ The agent chooses a series of (optimal) actions to achieve its goal.
e.g:- Tic Tac Toe action sequence.

★ Initial Simplifying Assumptions:-

① Fully observable.
② Deterministic
③ Static
④ Discrete
⑤ Known

★ Abstraction:- Remove irrelevant details to create a high level abstract representation.

e.g:-



★ Components of a Search algorithm:-

A search problem should have the following five components:-

① Start state

① Start State:-

First or root state to start a search algorithm.

② Actions:-

From any arbitrary state S_i, all actions from S_i could be listed.

③ Transition Model:-

The next state could be determined by choosing an action.

$$S_j = T(S_i, a_j)$$

④ Path Cost:-

- The cost of choosing an action.

⑤ Goal State:-

- The goal state is known.

State Space:-

State space is the set of all (reachable) connected states. So, the state space can be smaller than the possible number of states (unreachable)

* Do check examples in slides.

Vacuum world state space graph:-

S-I question → $n \times 2^n \Rightarrow 2 \times 2^2 = 8$

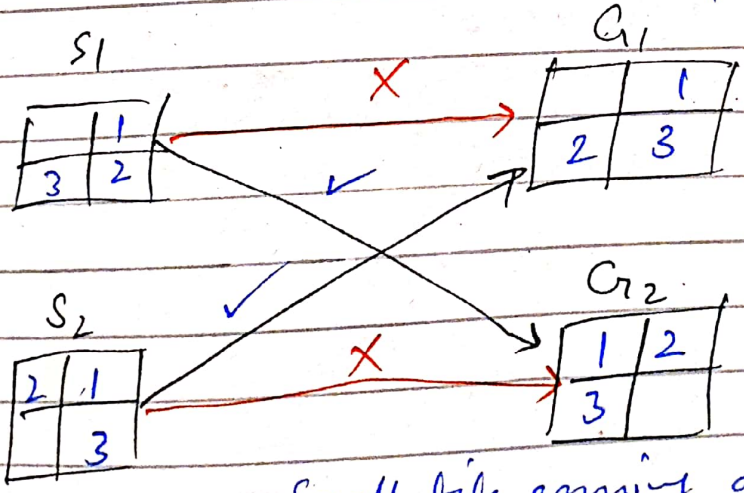
→ State changes based on the direction (states) & vacuum/agent location.

8-puzzle:-

State space why $\frac{9!}{2}$

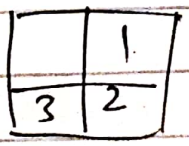
only even permutations are solvable

given $m \cdot n = 2$ $m \times n$ board



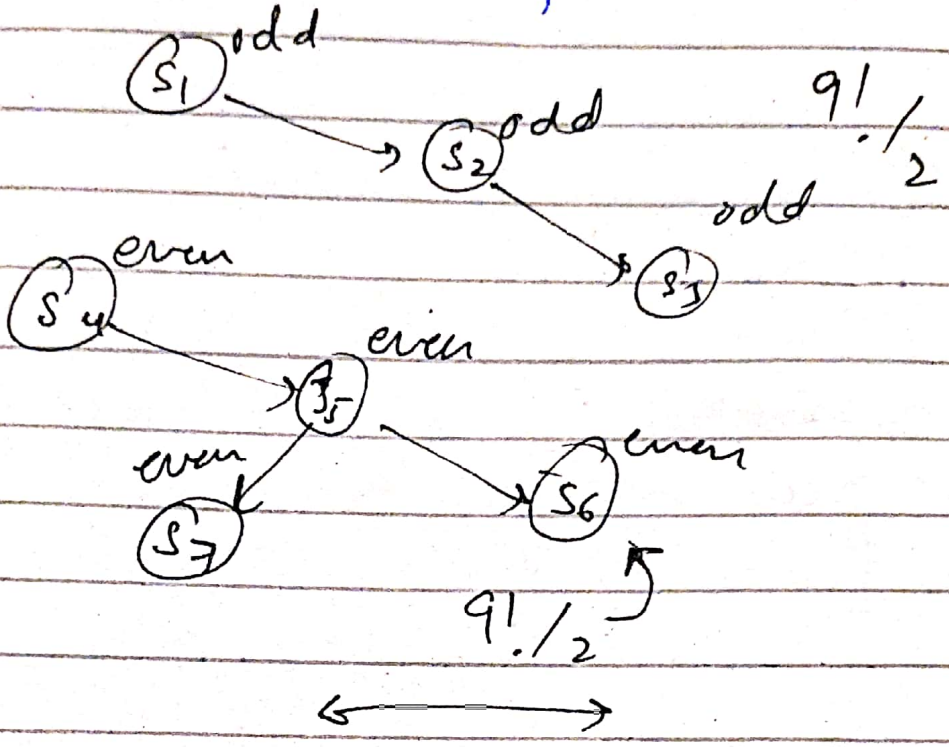
inversions = Small tile coming after
tile 1 + tile 2 + tile 3
0 + 0 + 1
⇒ 1 (Odd)

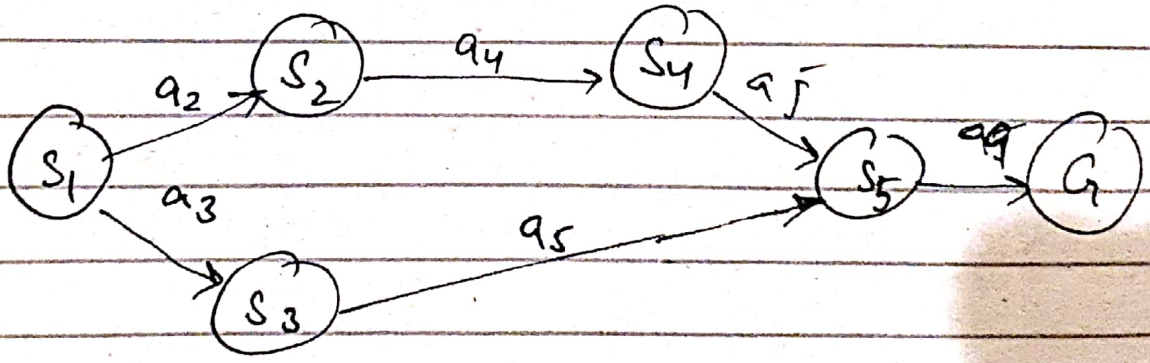
(only can be reached by odd parity start state).



Inversion in 8-puzzle:-

Inversion is the count of occurring tiles smaller than ith tile after it in 8-puzzle.





8-puzzle → Millirans

16-puzzle → Hundreds of Millions

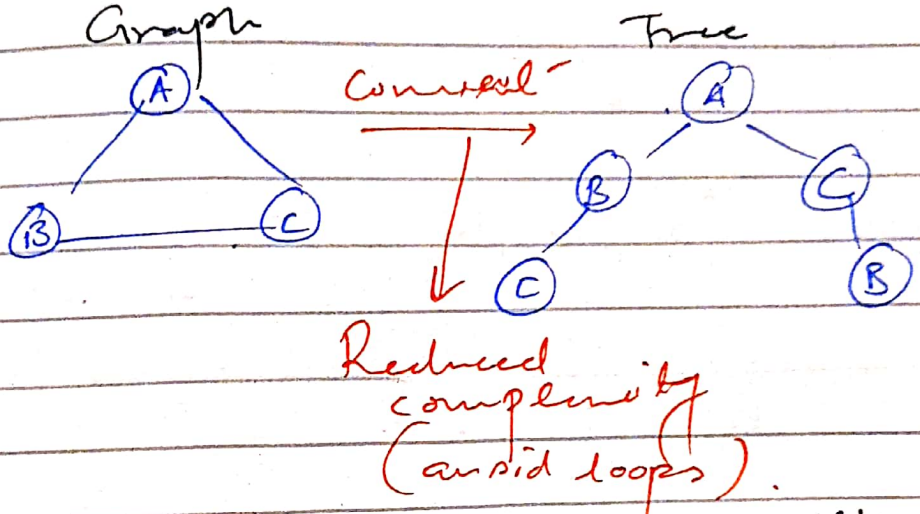
chess → 10^{120}

Vacuum Cleaner → $n \times 2^n$

* Not possible using conventional methods (Dijkstra).

Practical Solution--

→ Start with one state & keep expanding by maintaining a queue. Goal is to ~~Test~~ as few of states as possible. expand



Tree - Search Algorithms:-

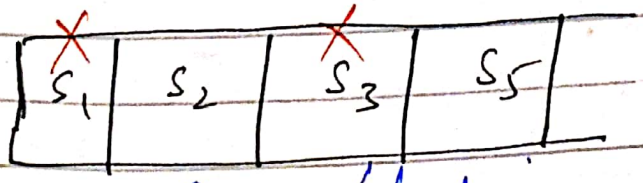
fn (problem, strategy) → Different types b/w algorithms.

→ returns a solution or failure

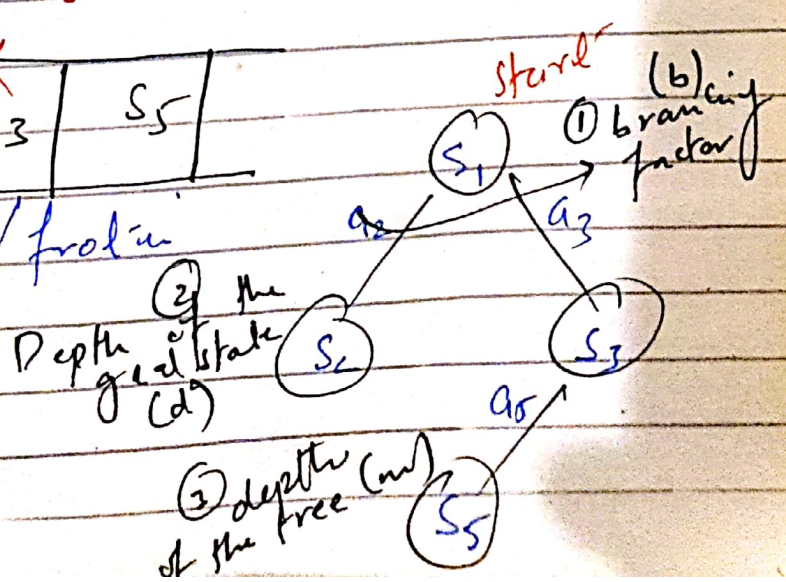
initialize the search tree using the initial state of problem.

⋮

→ In detail.



Queue / frontier



Differentiate b/w algorithms on the basis of :-

① Time Complexity :-
= The amount of time an algorithm takes to find a goal state usually represented in terms of number of nodes.

$O(b^d)$; $O(b^m)$

② Space Complexity :-
= The maximum # of nodes an algorithm stores in memory to find a solution.

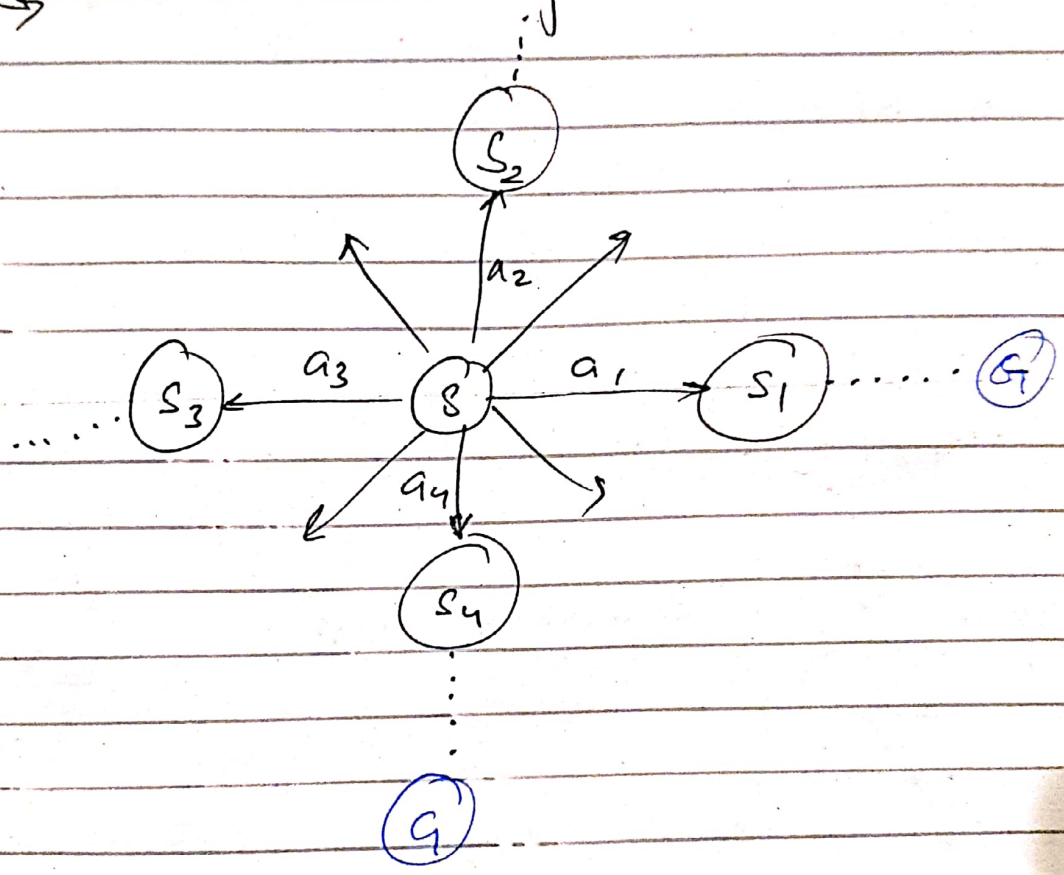
③ Completeness :-
= Algorithm should return a solution if one exists.

④ Optimality :-
= Should return least-cost solution (in case of multiple solutions).

✱ Uninformed Search Strategies :-

In which the algorithm only has search components & and other information about the state space.

→ 5 search components.



→ Such algorithms expand in all directions, Hence expensive.

→ Also, called blind searches.

Algorithms include :-

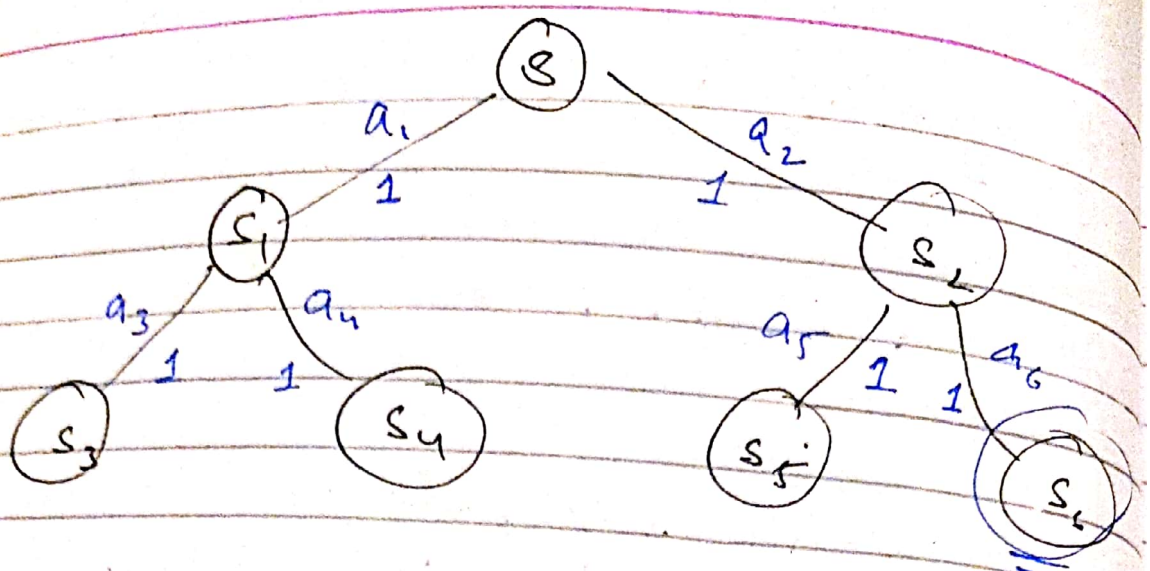
BFS → ① Breadth First Search

DFS → ② Depth " "

③ Uniform cost search

④ Iterative deepening search

⑤ Bi-directional search

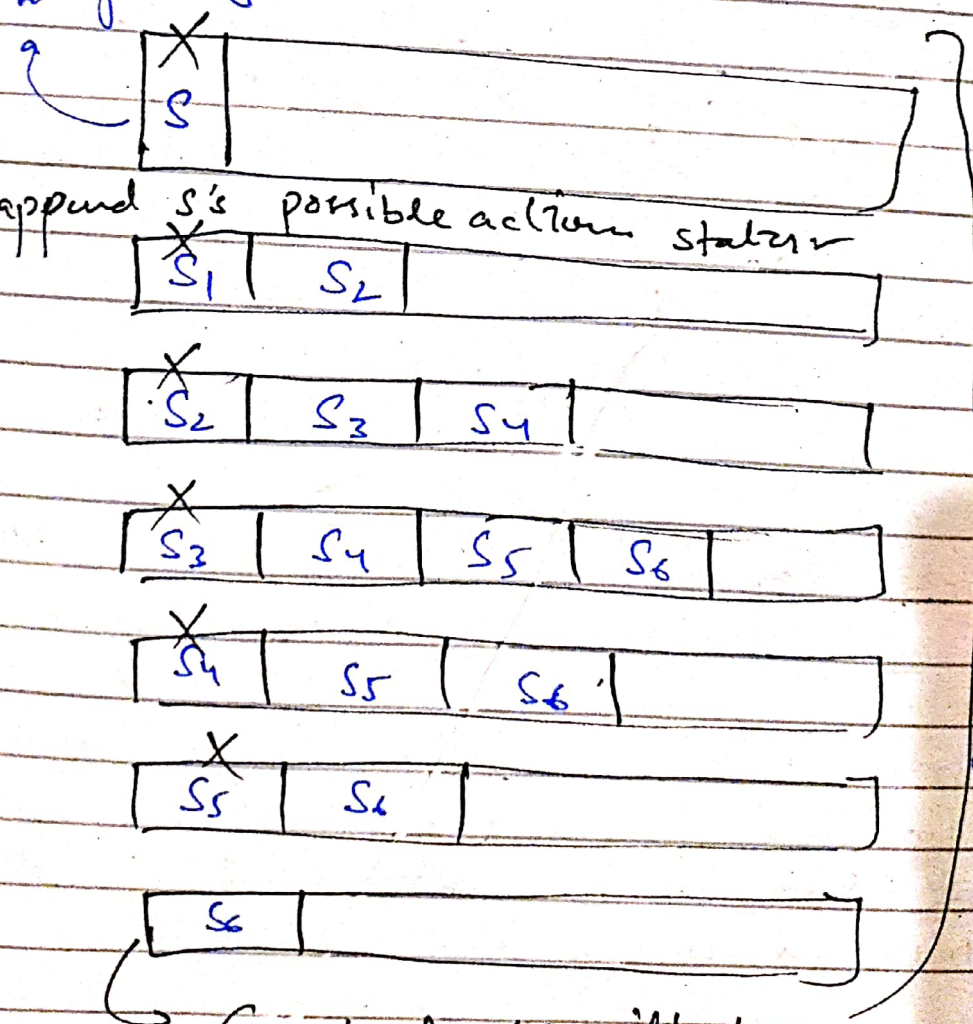


G_1

for BFS:-

This is the worst case scenario since BFS uses a queue (FIFO)

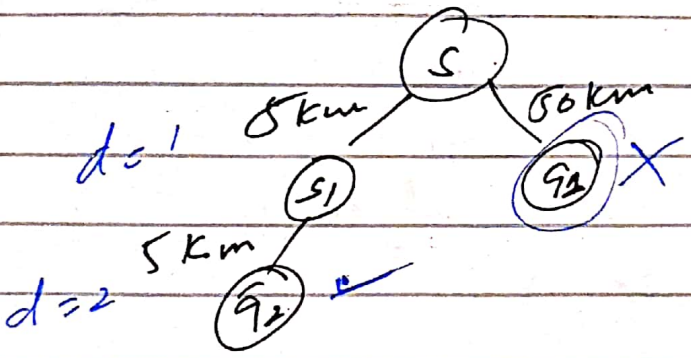
not goal (goal test) expend



→ Using this algorithm, you can generate the BFS - spanning tree (search space tree)

① Completeness → yes, if goal state is at finite steps / depth.

② Optimal → yes, if path cost is a non-decreasing ftn of depth.



③ Time Complexity :- depth →

$1 + 2 + 4$ (d=0, d=1, d=2)

$2^0 + 2^1 + 2^2$

$b^0 + b^1 + b^2$ ← branching factor

$b^0 + b^1 + \dots + b^d$

$O(b^d)$

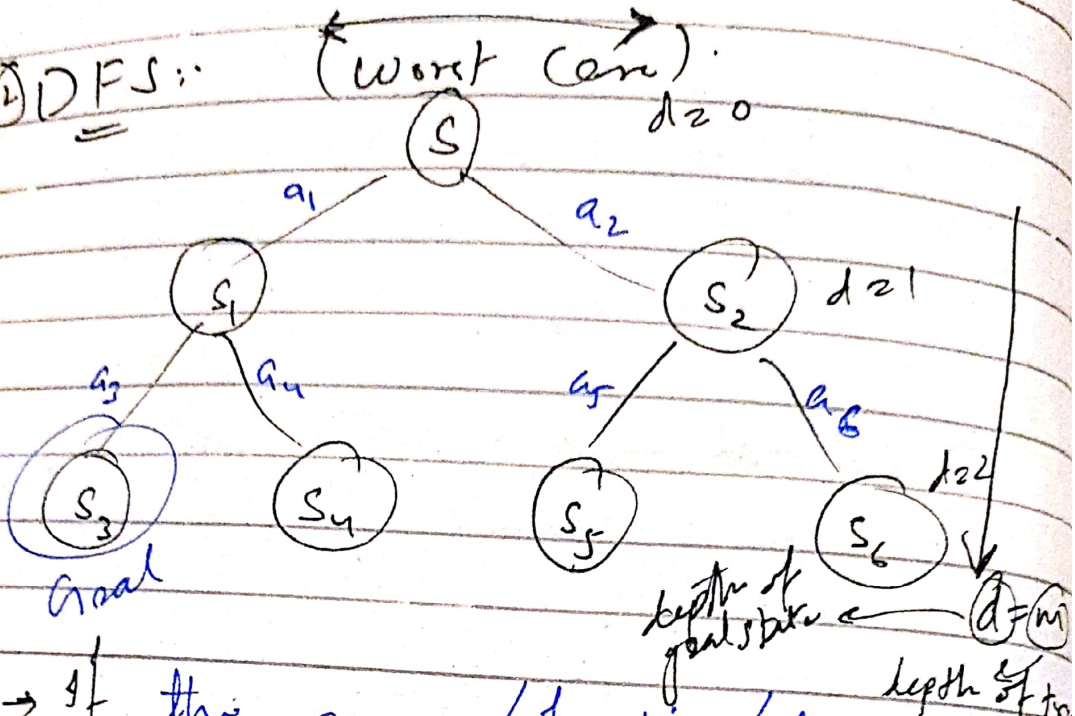
In terms of # of nodes.

④ Space Complexity :- b^{d+1}

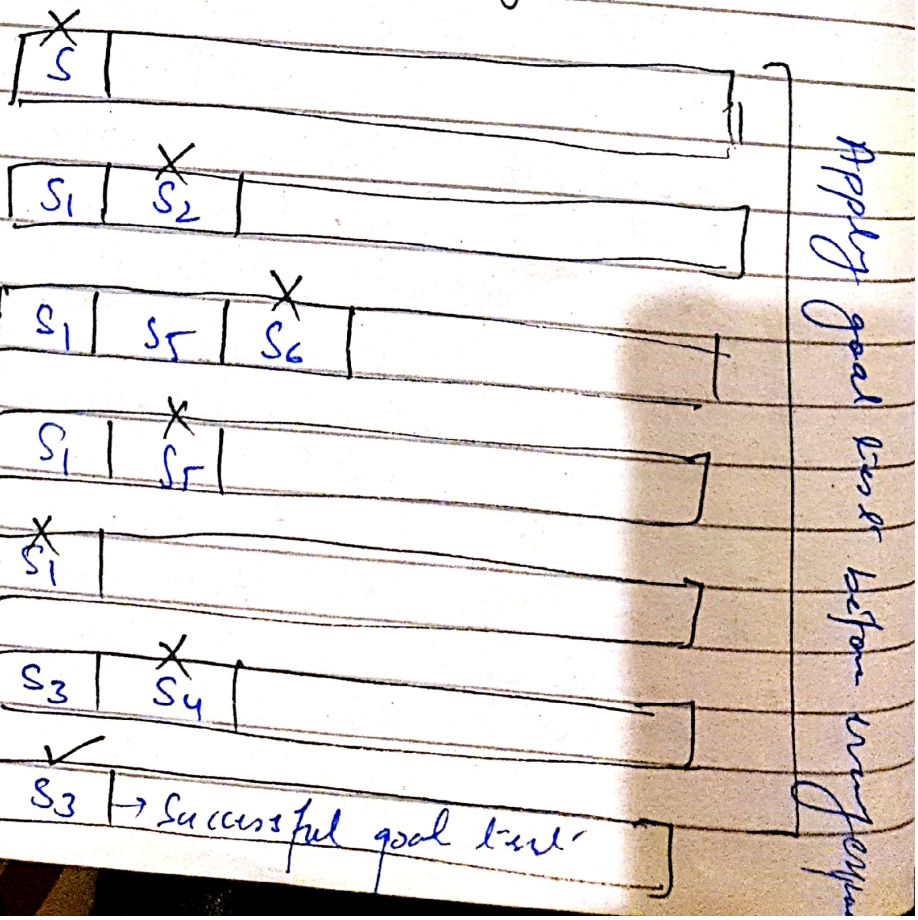
$O(b^d)$ //

When to use BFS? when the solution/goal is shallow i.e d is small.

① DFS:: (worst case).



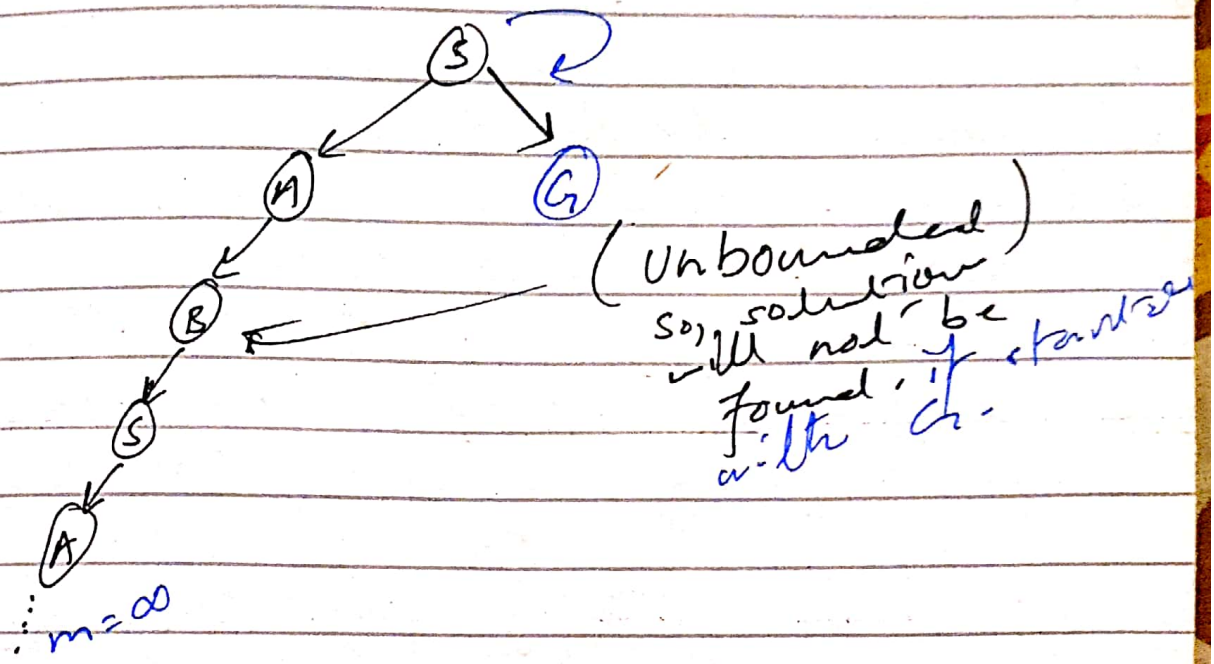
→ If the queue/frontier/fringe is switched from FIFO to LIFO (stack) then algorithm will become DFS.



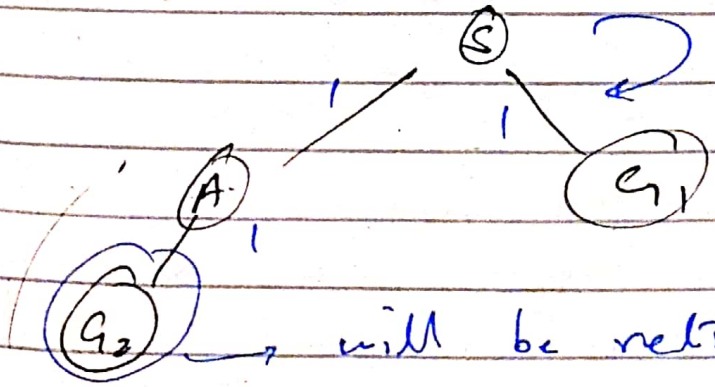
→ BFS expends shallowest node first vs. DFS expends deepest node ie.

① Completeness:- For trees with unbounded depth, completeness is false.

$m = \infty$



② Optimality:- is false, if suboptimal goal exists in the first deepest path.



→ will be returned.

(3) Time Complexity :-

$$\#\text{ of nodes} = \overset{m=0}{1} + \overset{m=1}{2} + \overset{m=2}{4} + \overset{m=3}{8} \cdots + \overset{m=x}{3_2}$$

$$= b^0 + b^1 + b^2 + b^3 + \cdots + b^m$$

$$= O(b^m) \quad //$$

(4) Space Complexity :-

$$\#\text{ of nodes} = \overset{d=0}{1} + \overset{d=m}{2} + 2 + 2 + 2 + \cdots + 2$$

$$= b^0 + b + b + b + b + \cdots + b$$

$$= 1 + m \times b$$

$$= O(bm) \quad //$$

$b = 10 \quad m = 12 \quad d = 5$

node = 1 Kbyte

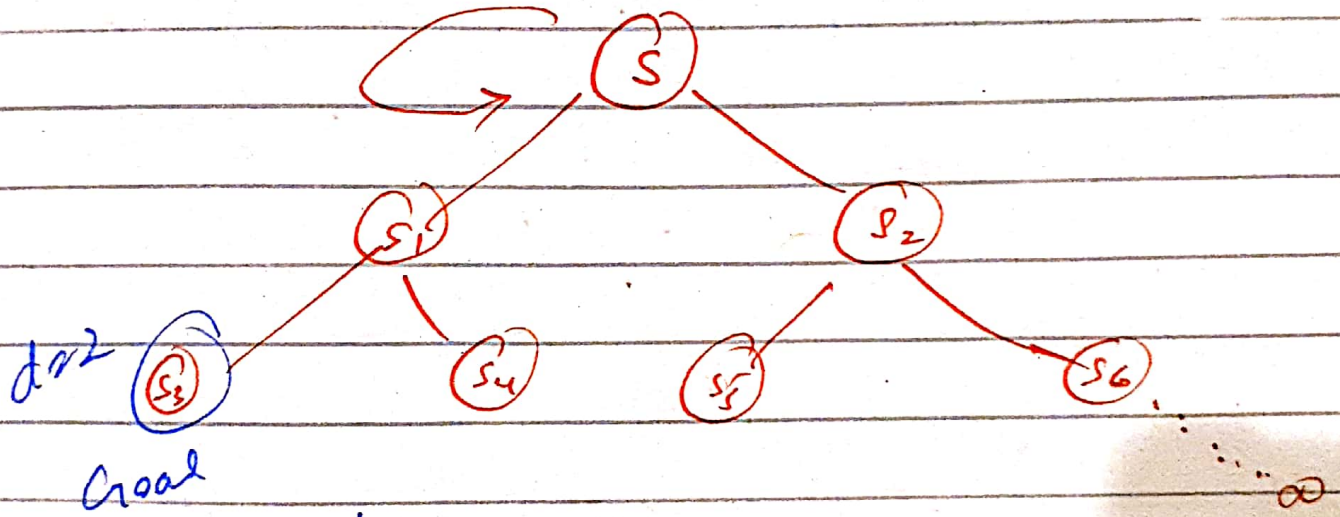
	Space Complexity	Time Complexity
$d=1$	120 Kb	10^{12} s
$d=2$	120 Kb	10^{12}
$d=3$	120 Kb	10^{12}
$d=4$	120 Kb	10^{12}
$d=5$	120 Kb	10^{12}

→ Iterative Deepening Search

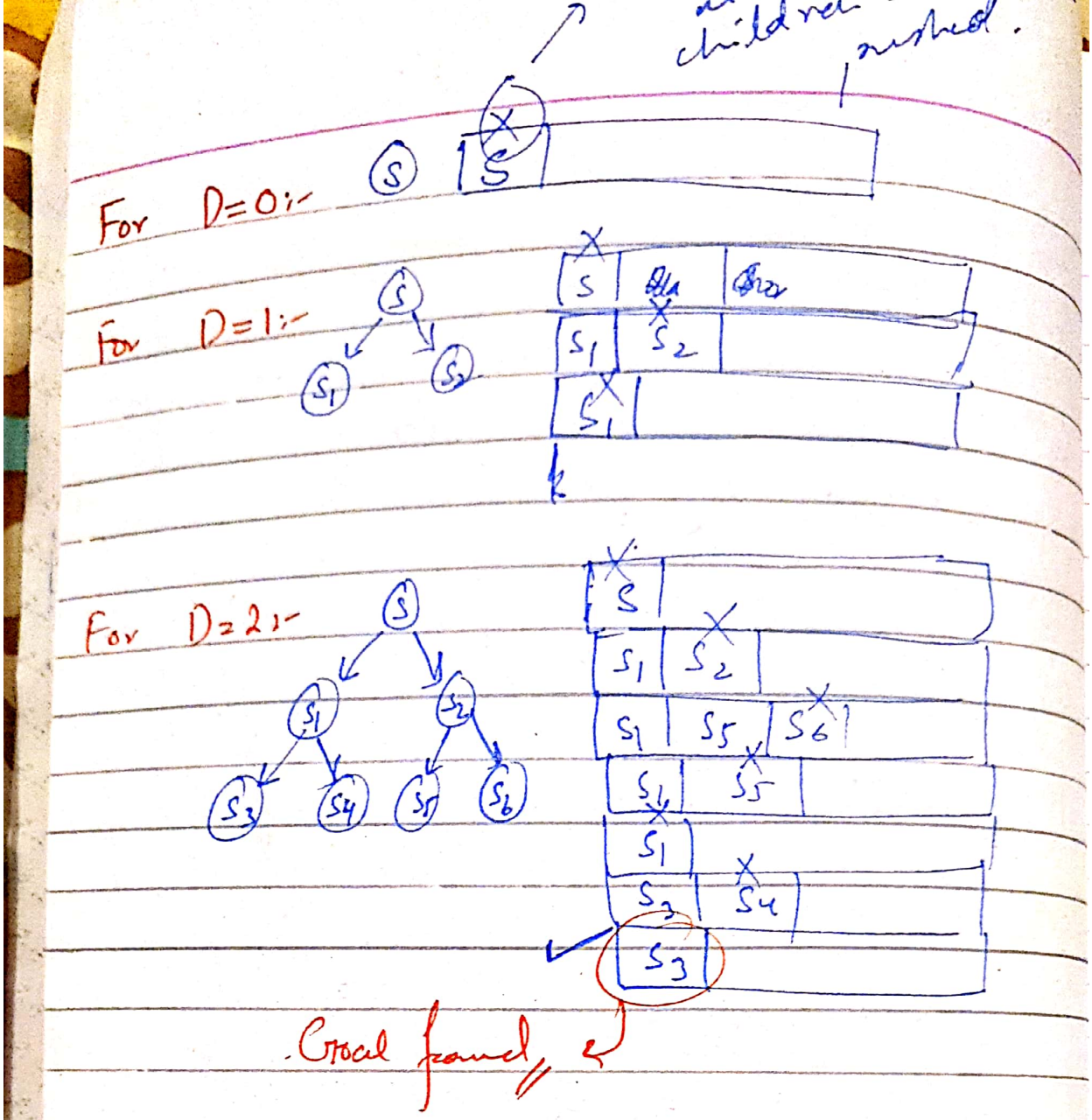
① Completeness Guarantees & optimality for problems with non-decreasing path cost.

② Have the space complexity of DFS i.e, linear!

③ slightly higher time complexity.




```
Solution = null;
D = 0;
while (solution == null) {
        solution = DFS (problem, D)
        D = D+1
}
```



(1) Completeness:- True, if solution/goal exists at finite steps.

(2) Optimality:- True, if non-decreasing path cost.

(3) Space Complexity:-

$$\#\text{ of nodes} = \sum_{d=0}^{d=\text{goal}} 1 + b + b + \dots + b^d$$

$$= 1 + b^d$$

$$= O(b^d)$$

(4) Time Complexity:-

Tree → $db^0 + b^1 + b^2 + \cdots + b^{d-1} + b^d$

$db^0 + (d-1)b^1 + (d-2)b^2 + \cdots + 2(b^{d-1}) + b^d$

$\Rightarrow db^0 + (d-1)b + (d-2)b^2 + \cdots + 2b^{d-1} + b^d$

Exact # of nodes

Complexity → $O(b^d)$

Comparison of Time Complexity:-

For $b=10$, $d=5$

BFS = 111,110 nodes

IDS = 123,450 nodes

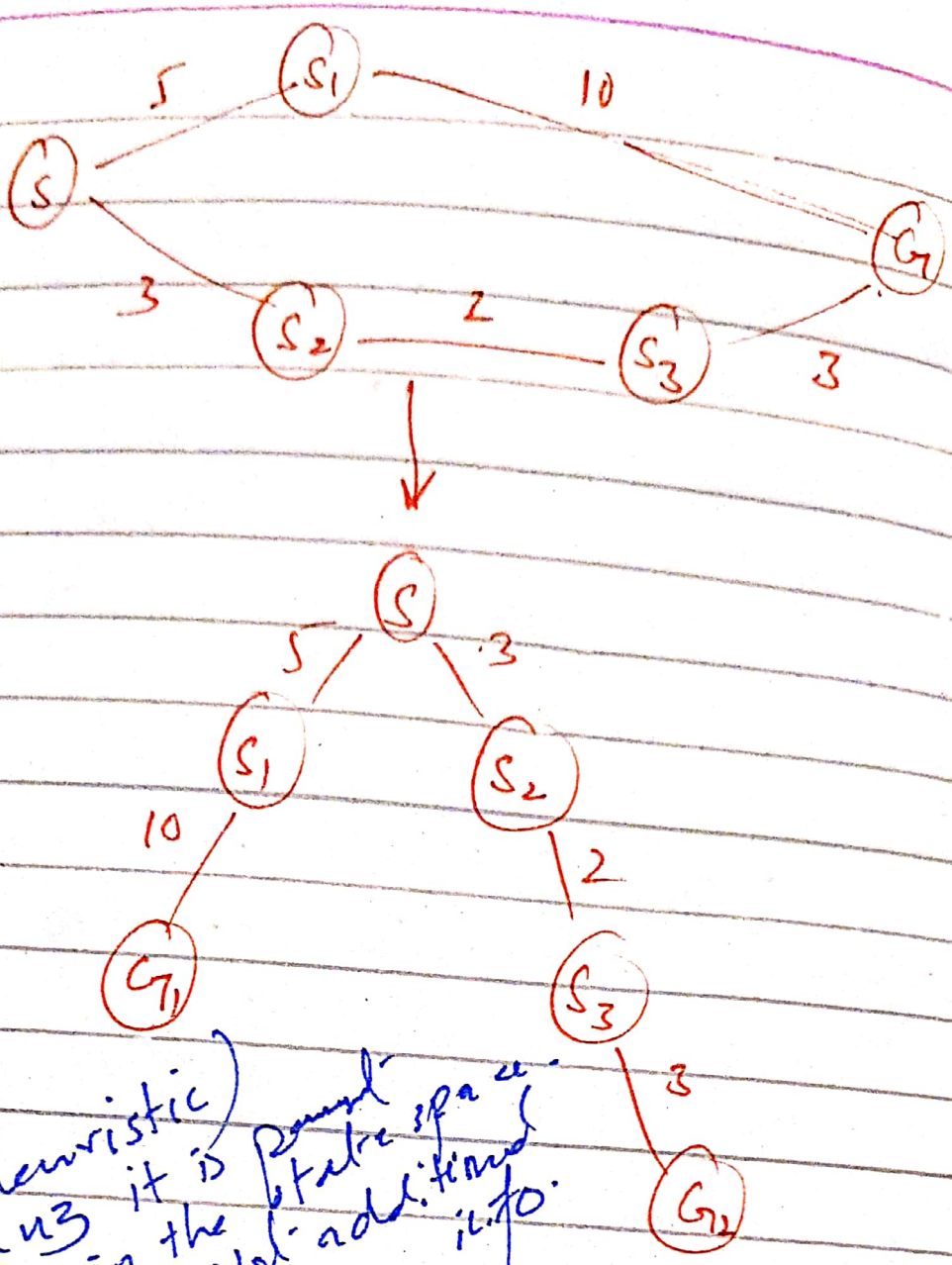
(Note the difference)

(4) Uniform Cost Search:-

(1) Suitable for problems with varying costs. (randomly distributed cost)

(2) Gives an optimal solution in such state spaces.

(3) The queue is prioritize at $g(n)$ cost, where $g(n)$ is the cost of node n from the root node (initial state).

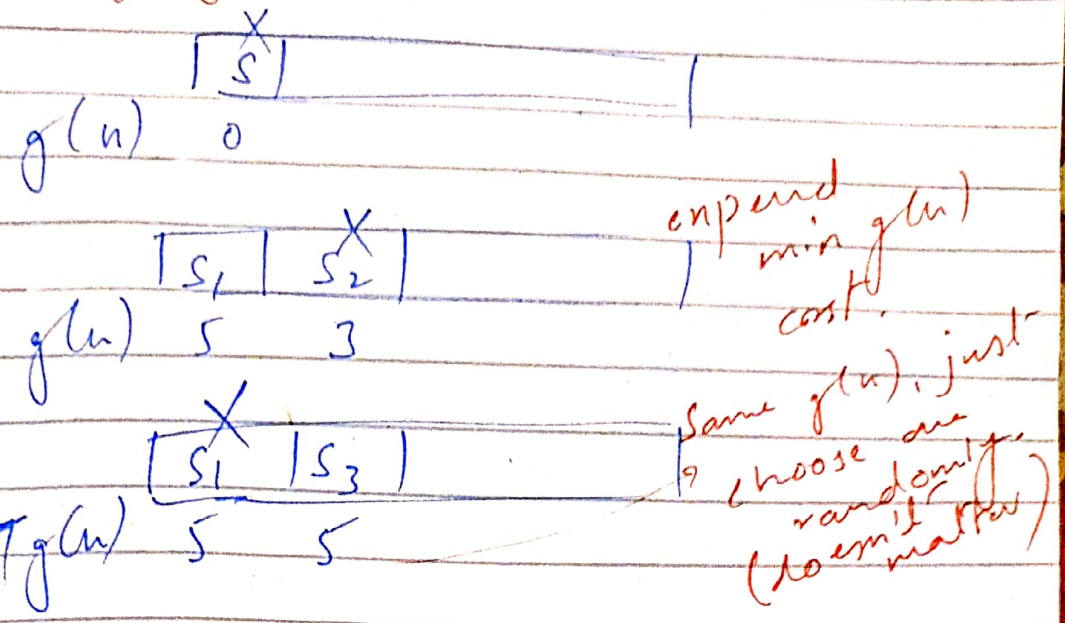


(Not heuristic)
cuz it is found in the state space & not added to it.

$g(n)$ = Cost of node n from root.

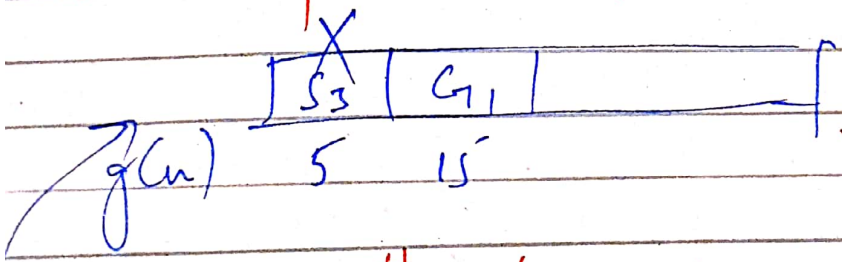
$g(S) = 0$
$g(S_1) = 5$
$g(S_2) = 3$
$g(S_3) = 5$
$g(G_1) = 15$
$g(G_2) = 8$ //

Priority g(n):-

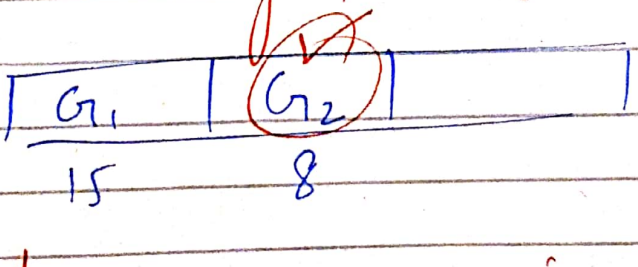


→ However, usually, we expand newer ones because we started from this path (optimal) so keep exploring it.

↳ Let's explore S1 (contradictory):-



→ even though we have the goal in the queue, we still expand the min g(n) cost.



→ Before every expansion, apply the goal test. So optimal solution:

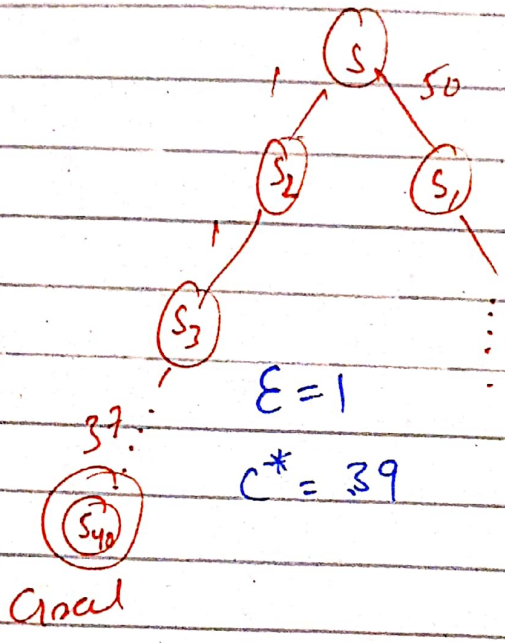
$G_2 \rightarrow S_3 \rightarrow S_2 \rightarrow S$

① Completeness:- Yes, if goal is at finite steps.

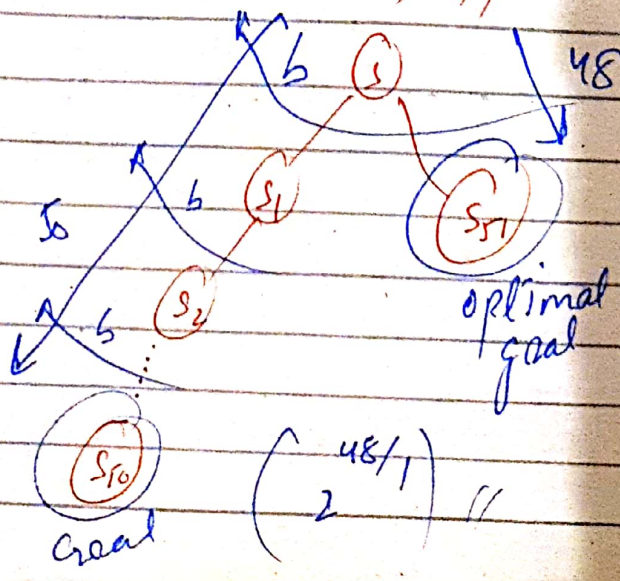
② Optimal:- Yes

③ Time Complexity:- C^* is the optimal goal cost.

ε is the minimum step cost.



$O(b^{C^*/\varepsilon})$ //



(5) Bi-directional :-

Actions are reversable & goal is a concrete state, not abstract.

→ Read yourself. //

CH#3 //

Informed Searches :-

* Heuristic fns
* Best first Searches
  i) ~~Uniform Cost Search~~ (Uninformed)
  ii) Greedy Best first Search
  iii) A* Search
  iv) Memory Bounded A*
  v) Recursive Best First Search.

→ Uninformed searches expand in all directions. If goal takes 30 steps with a branching factor of 4. So, 4^{30} nodes will be searched. which is expensive.

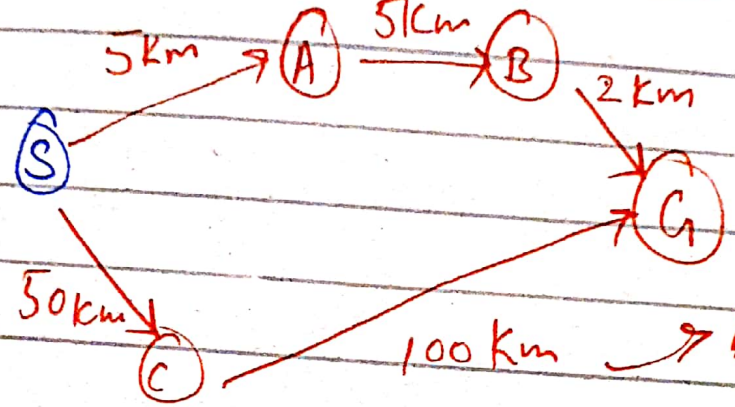
→ Additional information in informed searches will drive you to expand more in the goal's direction.

☆ Heuristic:-

① Rule of thumb to find a solution.

② Any additional information that speeds up the algorithm to find a solution.

Example:- Path finding Agent.



$h_1(n)$ = Straight line distance to goal state.

$h_1(S) = 6$ km
$h_1(A) = 4$ km
$h_1(B) = 1$ km
$h_1(C) = 60$ km
$h_1(G) = 0$ km

S.L.D < R.D

$\forall n, h(n) \leq h^*(n)$

Admissable Heuristic //

→ This is one heuristic, there can be multiple heuristics.

Another heuristic,

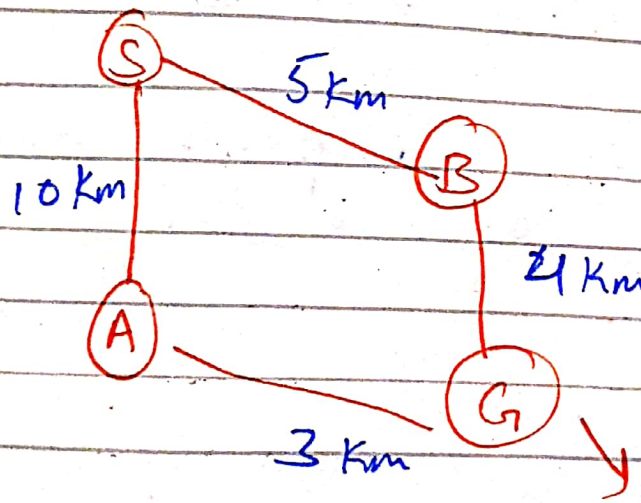
$h_2(n)$ = Railroad distances to goal state.

$h_3(n)$ = Aeroplane " "
" " .

$h_4(n)$ = Distance by foot.

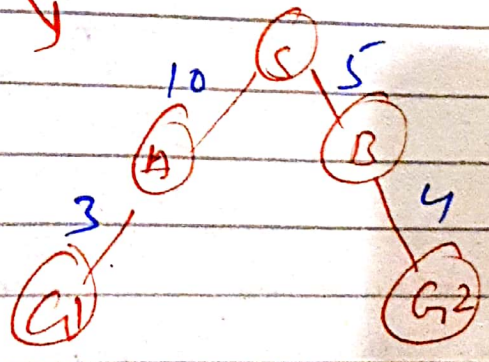
① Greedy Best First Search.

=

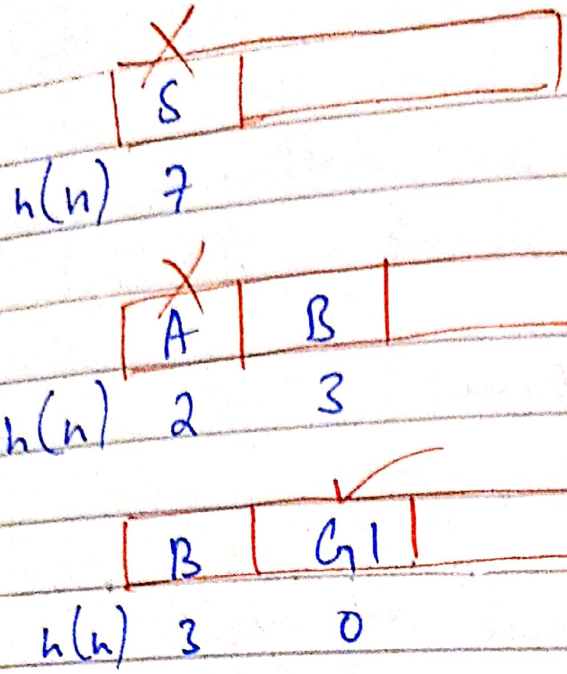


$h(n)$ = straight line distance.

$h(S) = 7$ km
$h(A) = 2$ km
$h(B) = 3$ km
$h(G) = 0$ km



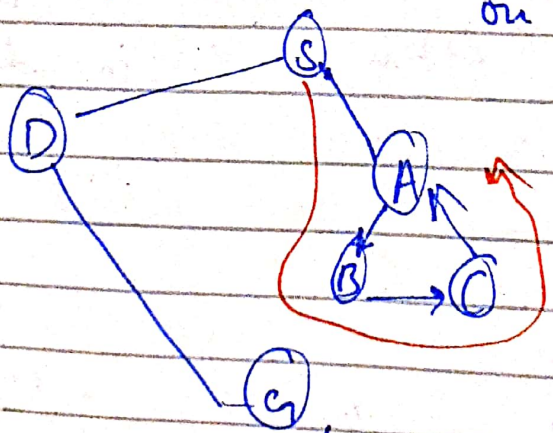
→ Queue, priority on $h(n)$ //



→ Goal is found but G1 is not the optimal solution.

① Optimality:- Not optimal.

② Completeness:- Not complete based on heuristic.



→ However, if loops are entered for, using trees then complete.

Double check this

③ Time Complexity :- $b^0 + b^1 + b^2 + \dots + b^m$

$= O(b^m)$ //

④ Space Complexity :-

$= O(b^m)$ //

1) Designing a heuristic function :-

2	3	1
4	5	6
7	8	

S_1

8	7	2
6	5	1
3	4	

S_2

1	2	3
4	5	6
7	8	

G

→ Assuming both are reachable. Which seems closer ? -

→ They seem closer, more # of tiles at the actual location in S_1.

$h_1(n)$ = # of misplaced tiles

↗ lower heuristic is better.

① We relax the rules / constraints of the game / environment.

e.g:- If move empty block in the misplaced tiles directly, then how many steps. i-e 3.

$h_2(n)$ →

$h_2(n)$ = manhattan distance of misplaced tiles from actual position in goal state.

$h_1(S_1) = 3$	$h_2(S_1) = 4$ (2+1+1+0+0+0...)
$h_1(S_2) = 7$	$h_2(S_2) = 18$ (3+1+4+2+0+2+4+3)

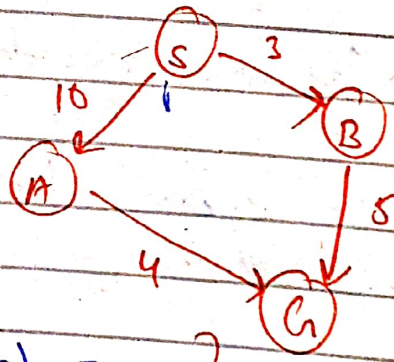
⟷

(11). A* Search:-

=

→ Queue prioritizes on $f(n)$, where

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



$g(S) = 0$ $h(S) = 7$
$g(A) = 10$ $h(A) = 3$
$g(B) = 3$ $h(B) = 4$
$g(G) = 8$ $h(G) = 0$

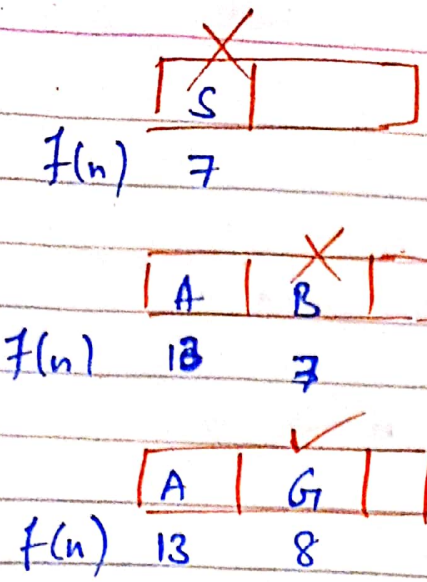
or convert into trees, to get both.

$h(n)$ = Straight line distance

$g(n)$ = Distance of n from root (S).

Admissible heuristic

$f(S) = 7$
$f(A) = 13$
$f(B) = 7$
$f(G) = 8$



Note:-
In case of ties, choose random or follow the same path.

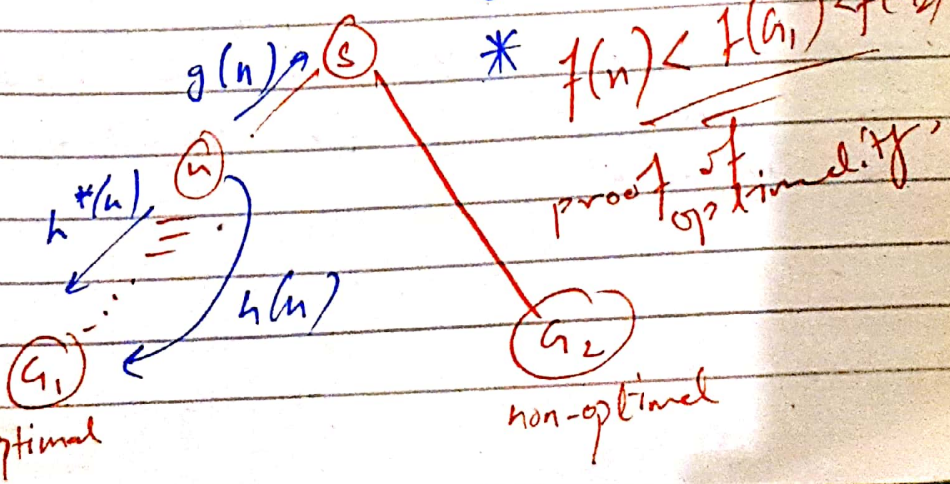
Optimal path:- S → B → G

→ Now, if you choose a non-admissable heuristic, it will not give optimal path.

① Optimality:- A^* is optimal, provided
For trees
i) $h(n)$ is admissable i.e $h(n)$ never overestimates the actual cost.

$$\forall n ; \underbrace{h(n)}_{\text{heuristic}} \leq \underbrace{h^*(n)}_{\text{actual cost}}$$

→ Admissability is for trees.



① $f(n) \leq f(G_1) \leq f(G_2)$:-

proof of optimality :-

$$f(n) = g(n) + h(n) \longrightarrow (a)$$

$$f(G_1) = g(G_1) + h(G_1) \quad //0$$

$$f(n) \leq g(n) + h^*(n)$$

. since $h(n) \leq h^*(n)$

$$f(n) \leq g(G_1)$$

$$f(n) \leq f(G_1) \quad //$$

if nodes are the same (ideal case)

$$f(G_1) = g(G_1) + 0$$

$$f(G_2) = g(G_2) + 0$$

(. By condition of optimality $g(G_1) < g(G_2)$)

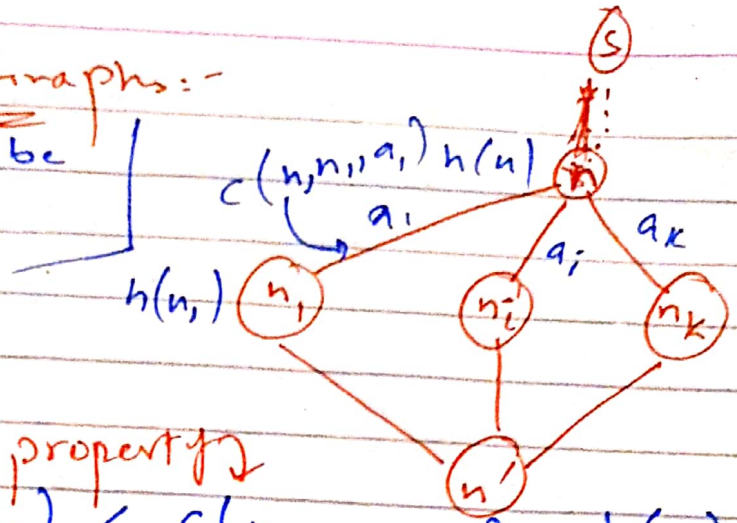
$$f(G_1) < f(G_2)$$

Proved $f(n) < f(G_1) < f(G_2)$ //

→ If the above statement holds for trees then, the solution will be optimal;

For Graphs:-

$h(n)$ should be consistent.



consistency property

$\rightarrow h(n) \leq c(n, n_i, a_i) + h(n_i)$

(II) $f(n) < f(n_i)$:-

proof of consistency

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

$f(n) < g(n) + c(n, n_i, a_i) + h(n_i)$ (By consistency property)

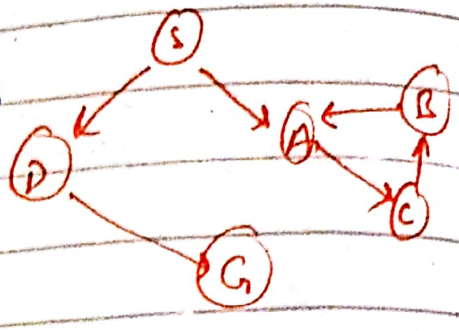
$f(n) < g(n_i) + h(n_i)$

$\boxed{f(n) < f(n_i)}$ // proved.

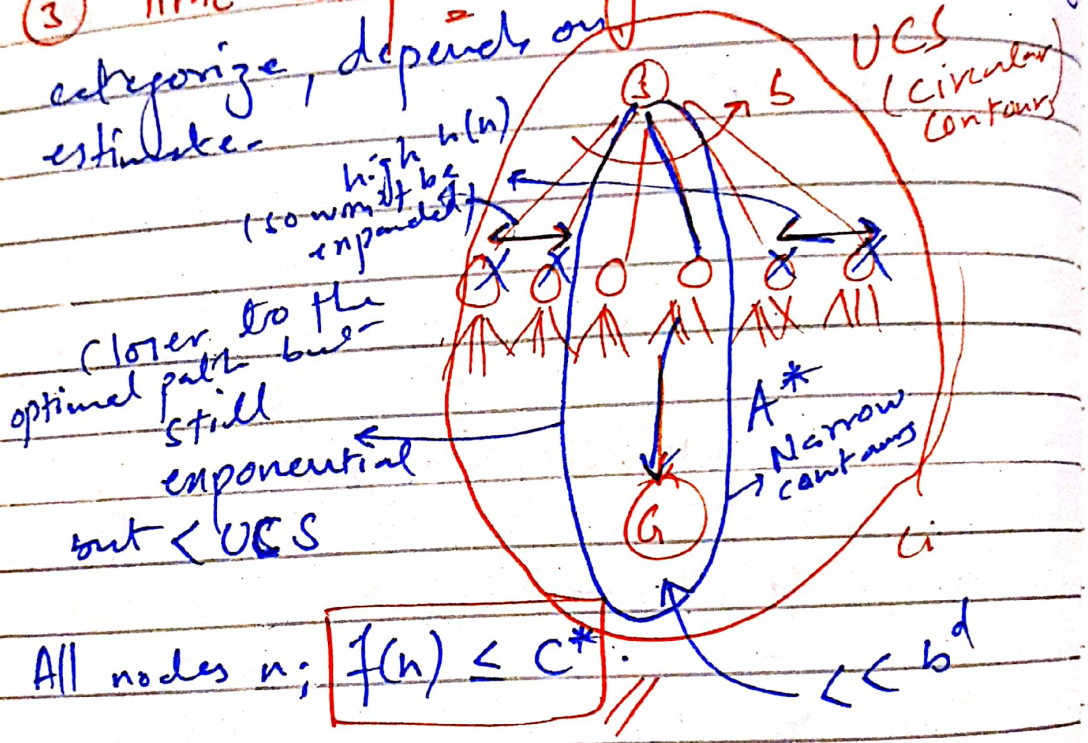
→ If the above statement holds for graphs for every node, then the solution will be optimal.

→ Given these 2 conditions, the solution will be optimal.

(2) Completeness:- Yes, even in such situations



(3) Time Complexity:- Cannot specifically categorize, depends on estimate.



(4) Space Complexity:- Much less than UCS.

→ mean of all admissible heuristics are the best. (maybe)

→ if h(n) = 0, then A* will become UCS.

(iii) Recursive Best First Search:-

i) A* search runs out of memory before running out of time.

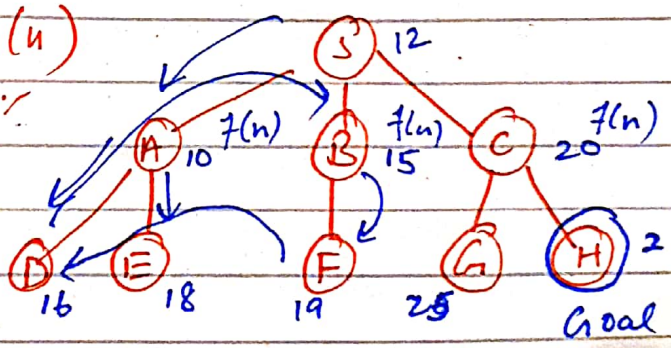
ii) RBFS is a memory efficient implementation of A* search.

iii) In RBFS Queue is priority on $f(n) = g(n) + h(n)$

iv) RBFS searches like DFS i-e only one depth at a time in a recursive fashion, using a variable called f-value.

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

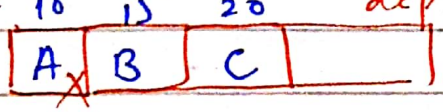
Space Complexity:
$O(bd)$



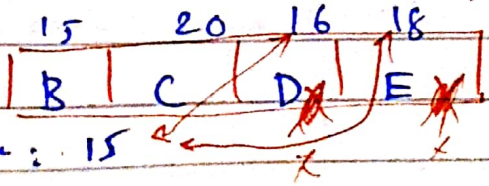
f(n)

f-value: ∞

f(n):-



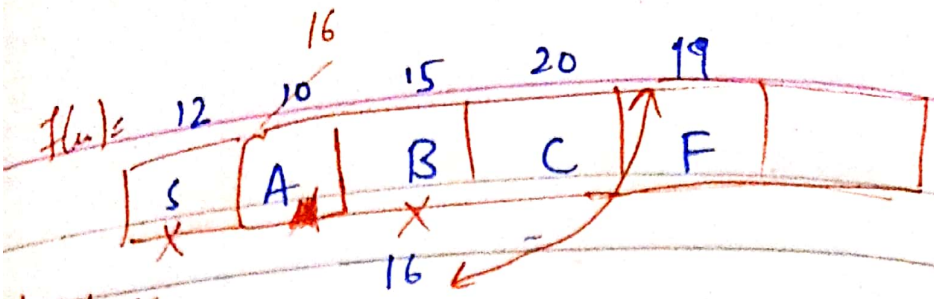
f-value: 15



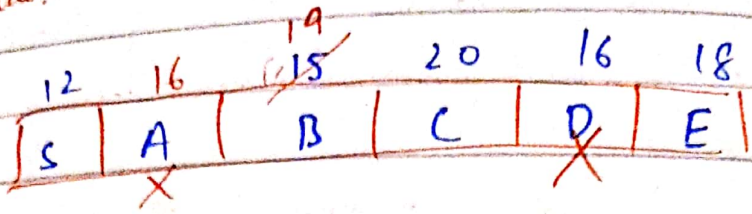
f-value: 15

max 1 depth nodes in queue at a time

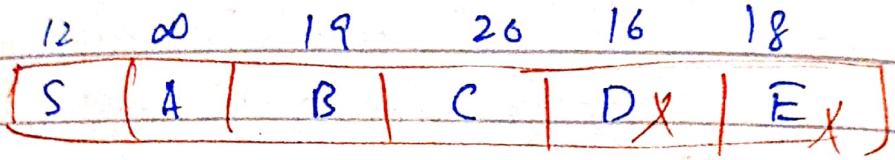
* f-value is the alternate path cost.
→ the moment f(n) exceeds f-value, algo backtracks.



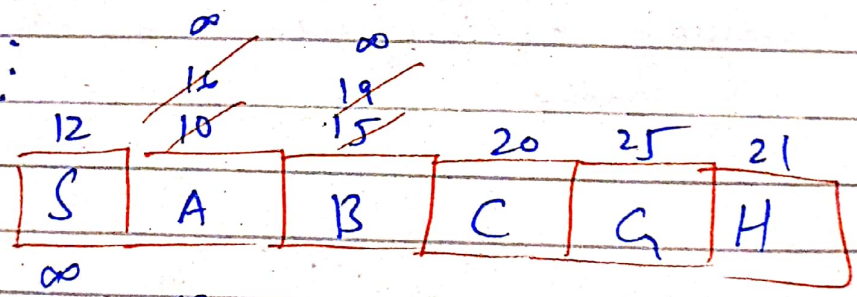
f-value:-



f-value:- 19 18



f-value 19



f-value:-

→ See Ramnic example for better understanding //

Optimality:- Yes, given the 2 conditions of A*.

Time:- Difficult to characterize Depends on h(w).

Notion of dominance:-

If $h_2(n) \geq h_1(n)$ for all n (both admissable) then h_2 dominates h_1 aka. h_2 is better for search.

Effective branching factor:- (b^*)

$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

→ ~~Measure~~ provides a good guide to the heuristic's overall usefulness.

Chapter # 04:-

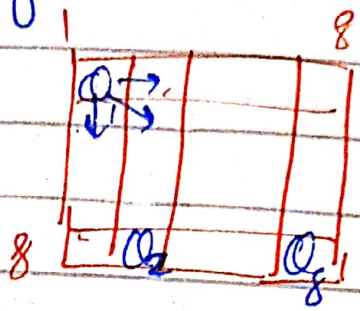
Local Search & Optimization

1. Problem Formulation.
2. Hill climbing & variants.
3. Simulated Annealing Algorithm
4. Genetic Algorithms

① Problem Formulation & Related Concepts

i) In local searches the path to goal state is not important & we are only interested in goal state.

8-queen problem →



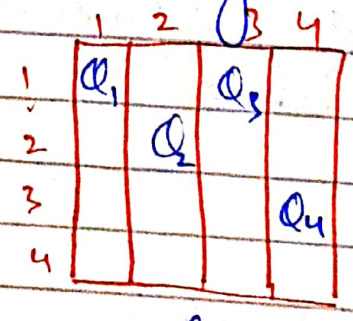
(12 goals)

ii) A state in local searches has a fitness value (maximization problems) or objective value (minimization problems).

fitness function → heuristic

objective function.

iii) In local searches we search in the local neighborhood & ignore the previous state after taking an action.



S

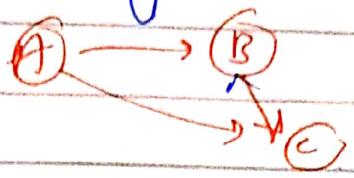
Fitness fn. (for every state)
f(n) = # of non-attacking queen pairs

→ state aka solution / individual / ...

Notes:-

* → for travelling salesman, objective fn can be:-

f(n)+g(n) = Distance by visiting each city at most once.



A B C
A B C B A
....

Back to Queen problem ⟷

$f(q) = {}^4C_2 \rightarrow$ (no. of ways pairs)

(Goal) = $\frac{4!}{2! \times 2!} = 6$

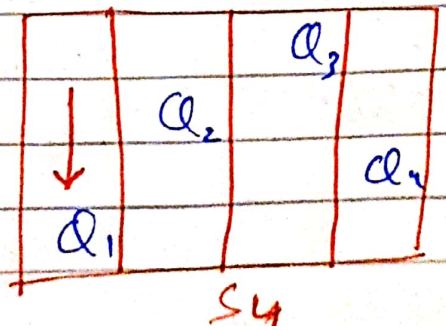
q_1	q_2
q_1	q_3
q_1	q_4
q_2	q_3
q_2	q_4
q_3	q_4

$f(s) = 3$ (Our state)

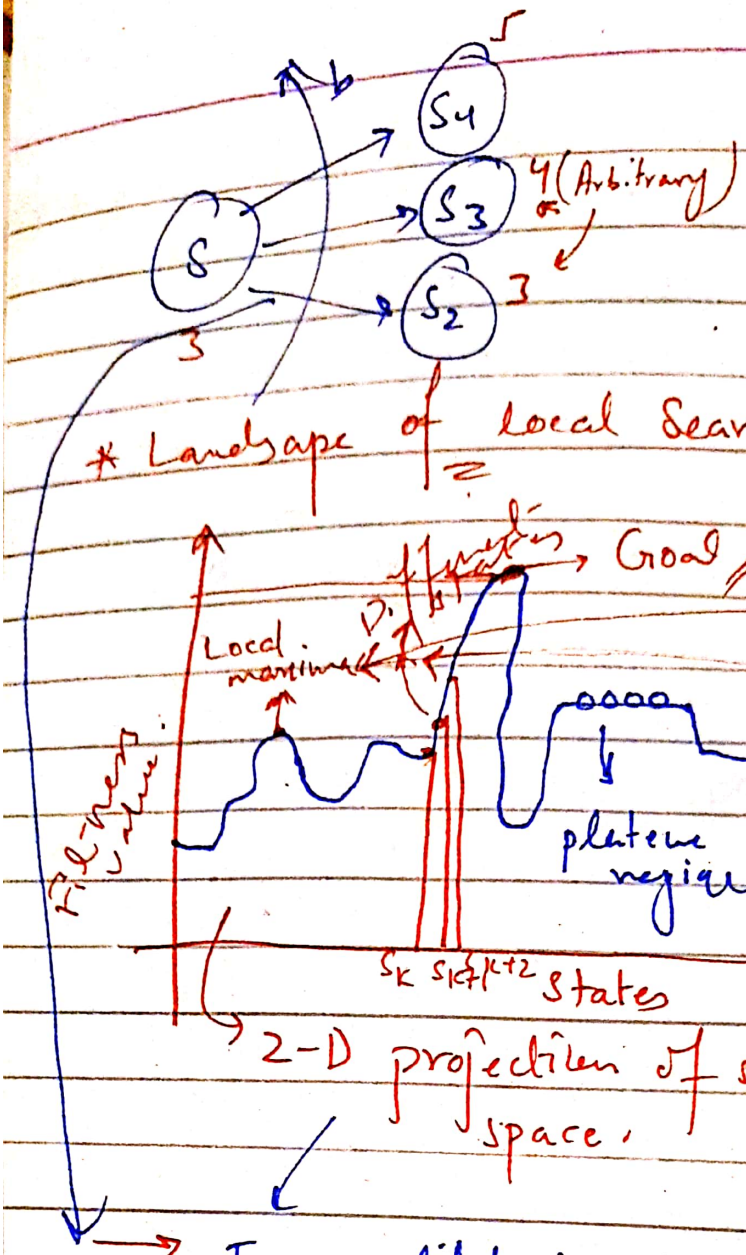
q_1	q_2	✓
q_1	q_3	✓
q_1	q_4	X → 3
q_2	q_3	✓
q_2	q_4	X
q_3	q_4	X

actions:- move queen q_i to any location (causes state change)

eg:-



$f(s_4) = 5$



→ In reality, it depends on the branching factor. Using 2-D graph for simplicity.

need b+1 dimensional space.

Three types of problems:-

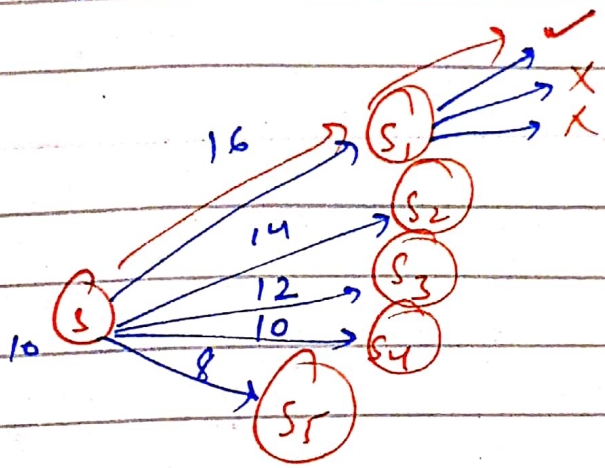
① Local maxima:- Algorithm is stuck in local maxima. (Local region of search landscape having the maximum value but less than global). e.g. 10^{80} states >>

(2) Ridges: Series of local maxima connected together.

(3) Plateau Region: A region where fitness fn is flat.

(1) Hill climbing Algorithm:-

→ Choose the highest fitness state from the current state.



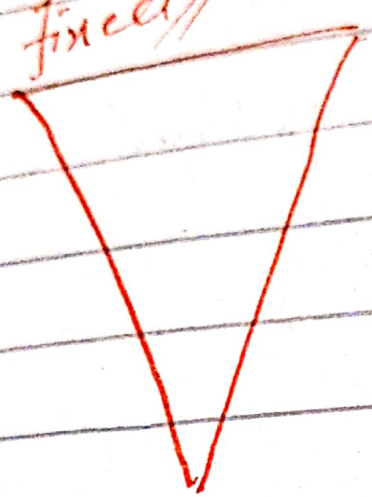
Now gets stuck at the local maxima,

Problem with hill climbing :→ ridges & plateaus.

→ So, solution to this problem is resetting using randomness when gets stuck.

Simulated Annealing → Physical randomness
Genetic → Biological randomness
Ant colony →
Swarm → : Each have their own way of achieving randomness.

Simulated Annealing



High T → high prob

T = 1000 ; P = 1

T = 1 ; P =

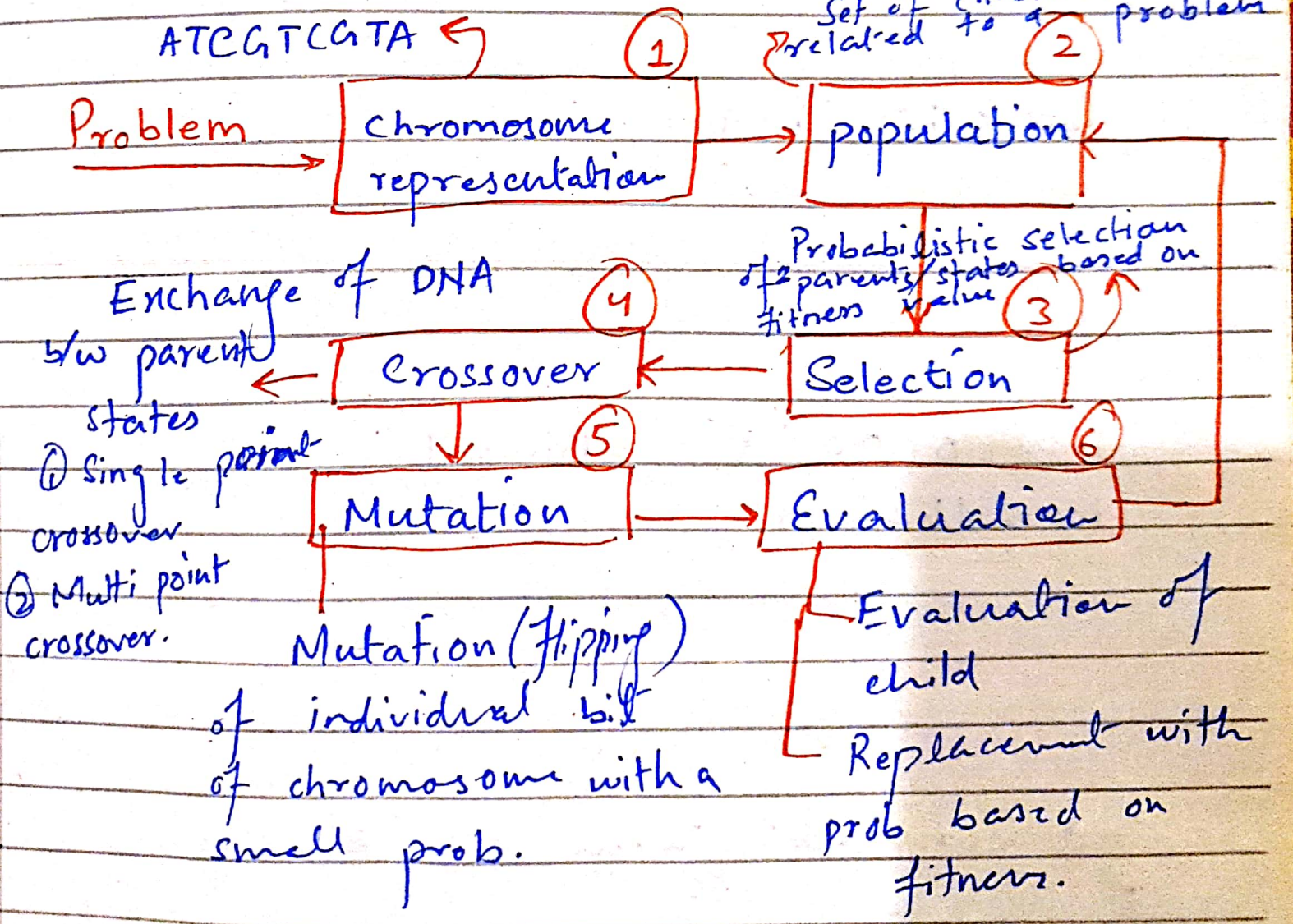
Low T → Low prob

$P_i = e^{\Delta E / T}$ → Increase in T will not decrease P_i because ΔE is negative.

✡ Genetic Algorithm :

→ Randomness inspired from biological evolution.

→ Parallel search algorithm based on sexual reproduction, i-e next state will be generated from two parents.



3x3 image

0	0	0
0	0	0
0	0	0

start

3x3 image

1	1	1
1	1	1
1	1	1

Goal

$f(n)$ = The # of 1s in a state n.
$f(n)$ = The # of 0s in a state n.

Chromosome Representation:-

↳ ATCGTCGTA

→ This can be represented in multiple ways.

① 1 0 1 0 1 0 0 1 0 (Binary)

② 1 2 3 4 1 2 3 4 1

A – 1
T – 2
C – 3
G – 4

→ Convert from 2D to 1D.

1 2 3 4 5 6 7 8 9
000 000 000

1	2	3
4	5	6
7	8	9

→ Every chromosome is a solution & has a fitness value

$f(\text{start}) = 0$

→ Also given a chromosome, can be converted into a state.

S_1 010 . 000 100

$f(S_1) = 2$

2. Population:-

Generate individuals randomly.

Randomly generated chromosomes | Population

P_1	000	100	000	$f(n)$ = # of 1's.
P_2	100	001	010	
P_3	001	000	100	
P_4	000	110	000	

Population ↑ = Diversity

↓ = No diversity

→ Get their fitness value.

$f(P_1) = 1$, $f(P_2) = 3$, $f(P_3) = 2$, $f(P_4) = 2$

3. Selection:-

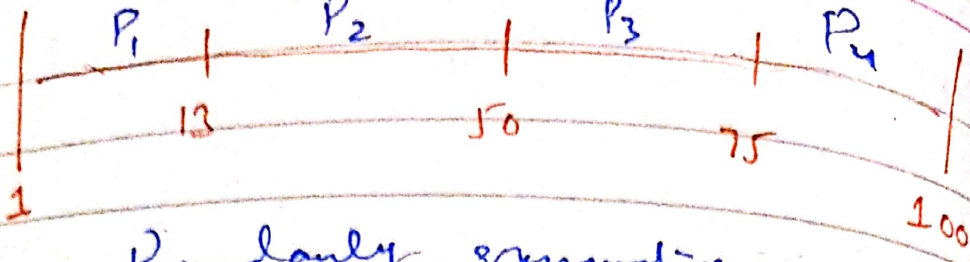
$$P_i = \frac{f(P_i)}{\sum_{j}^{\text{population}} f(P_j)}$$

$P_1 = 1/8 = 0.125 = \frac{12.5}{100} \approx 13/100$

$P_2 = 3/8 = 0.375 = 37.5/100 \approx 37/100$

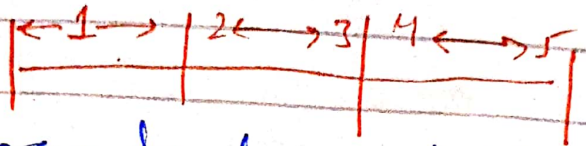
$P_3 = 2/8 = 0.25 = 25/100 \approx$

$P_4 = 2/8 = 0.25 = 25/100 \approx$



→ Randomly generate number
eg got 40, so we select P_2.

Left are: P_1, P_3, P_4
1/5 2/5 2/5



→ So randomly select P_2's pair, which can be either one of them.

① P_2 — P_3
② P_1 — P_2
⋮

④ Crossover:
Single point crossover

P_2 100 001 | 010
P_3 001 000 | 100
e.g ⑥

→ Generate random number to seperate the P.

s/w 1-4

C_1 100 001 100
C_2 001 000 010

for :- P_1 & P_2 :- of (4)

P_1 000 | 100 000
P_2 100 | 001 010

C_3 000 | 001 010
C_4 100 000 000

(5) Mutation :-

$P_m = 0.01$

Out of 100 bits → 1 will be mutated.

eg :-
C_1 becomes
101 101 100

(6) Evaluation :-

	$f(c_i)$	$f(g) = 9$
C_1	5	→ check for goal state, here
C_2	2	
C_3	3	
C_4	1	

→ If the goal is found, stop. Else replace the children with population based on the fitness of the population. (loop)

→ If f(a) is not known, keep doing it until f() stops increasing.

2 imp steps → Crossover & mutation

→ because crossover is convergence operator

→ mutation is exploration operator.