Artificial Intelligence

Raslan Kiwan

Chapter 1: Introduction

* Rule Base system (RBS):

RBS components are: Working memory, Rule Base, interpreter.

(1) Working memory: (WM)

- Contains facts about the world observed
- Contains temporary knowledge about problem solving
- Can be modified by the rules.

(2) Rule Base:

- Contains rules, each rule is a step in problem solving
- Rules are domain knowledge and modified only from outside (domain)
- If the conditions are matched to the working memory, then rule will be fired.

(3) Interpreter:

- It is the reasoning mechanism in RBS
- It operates in cycles
  * retrieval: find rules that match the current WM
  * Execution: execute the action of the rule.

Chapter 2: Problem solving and Search Strategy:

Main goal: Automatically solve a problem.

to do this; we need:

① representation to the problem.

② Algorithm uses strategy in order to solve the problem in that representation.

* Representation:

full description for a problem:

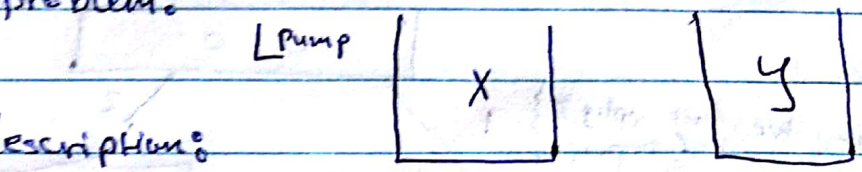
- State representation:
  - set of states (State space).
- Initial state. } → special states
- Goal state }
- set of actions (operators):
  - successor functions: function used to move from a state to another.
- path cost

---

* Tree Vs Graph: Non linear data structure

Nodes	No
Connected by rules	Rules in connections
N-nodes, N-1 edges	

* Water Jug problem:



* Problem description:

- State representation: Pair of integers. (x, y)

  $0 \leq x \leq 4, \quad 0 \geq y \leq 3$

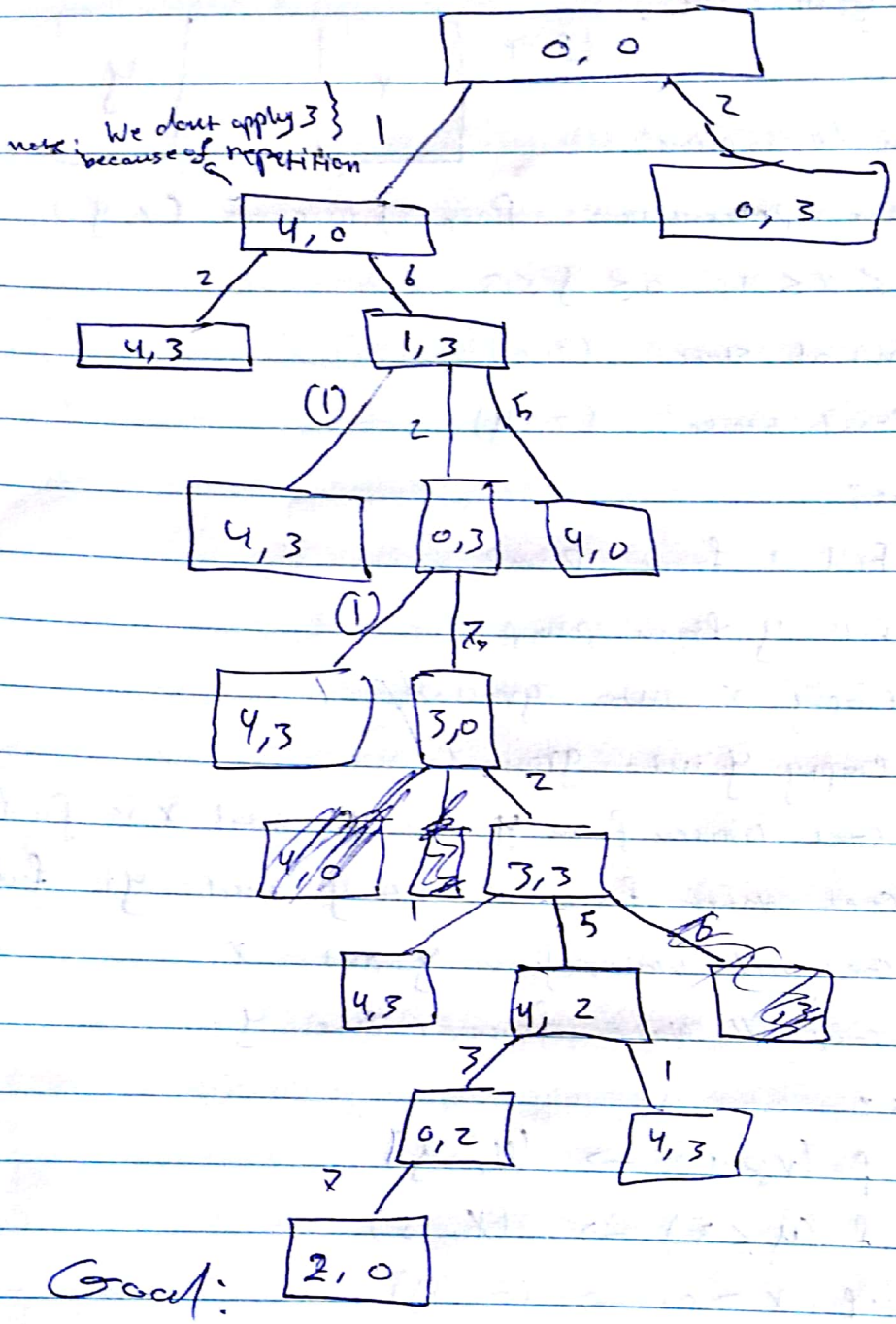
- Initial state: (0, 0)
- Goal state: (2, y)

* Actions:

1 - Fill x from pump
2 - Fill y from pump
3 - Empty x into ground
4 - Empty y into ground.
5 - Get water from y into x until x is full
6 - Get water from x into y until y is full
7 - Get all water from y into x
8 - Get all water from x into y.

* Rules:

(1) if $(x < 4) \rightarrow (4, y)$
(2) if $(y \leq 3) \rightarrow (x, 3)$
(3) if $(x > 0) \rightarrow (0, y)$
(4) if $(y > 0) \rightarrow (x, 0)$
(5) if $(y > 0)$ && $(x+y \geq 4) \rightarrow (4, (y+x-4))$
(6) if $(x > 0)$ && $(x+y \geq 3) \rightarrow ((x+y-3), 3)$
(7) If $(y > 0)$ && $(x+y <= 4) \rightarrow (x+y, 0)$
(8) if $(x > 0,$ && $x+y <= 3) \rightarrow (0, x+y)$



* Algorithm for solving a Problem:

1- Initialize the search tree using the initial state of the problem.
2- Choose a terminal node for expansion according to search strategy
   - if there is no terminal node for expansion return failure.
   - If the choosen node was the goal, return node.
3- Expand the chosen node (according to the rules) and add the successor to the search tree (list).
4- Go to step 2.

---

* Missionaries & Cannibals Problem:

* Problem description:
   - State representation:
     (ML, MR, CL, CR, B)
     B: boat; 0 on left, 1 right.
   - Initial state: (3, 3, 0, 0, 0)
   - Goal state: (0, 0, 1, 3, 3)

* Actions:

1- carry (2, 0) (M, C)

2- carry (1, 0)

3- carry (1, 1)

4- carry (0, 2)

5- carry (0, 1)

* Rules:

1- We ~~can one~~ can move one missionary in one side (M >= 1 && (M-1 >= C || M-1 = 0)

in other side M+1 >= C

2- Two missionaries:

can move if, in one side:

M >= 2 && (M-2 >= C || M-2 = 0)

3- One missionary, one cannibal

can move if, in one side M >= 1 && C >= 1

in other side, M+1 >= C+1

4- one cannibal can move if in one side:

C >= 1 && (M >= C-1)

in other side: (M >= C+1) || M = 0

5- Two cannibals can move if

in one side: (C >= 2) && (M >= C-2)

in otherside: (M >= C+2) || M = 0

* Note: Any Problem can be solved By using rules combination with control strategy

* Search:

* Breadth first search (BFS):

is it complete? Yes

Is it optimal? No, takes shallowest solution

* Averag branch factor in a tree: $\frac{\#\text{nodes}}{\#\text{levels}}$

$\#\text{nodes} = \sum b^d$

b: branch factor, d: depth of each level

* Big O: (worst case)

* Time complexity: $O(b^d)$

* space complexity: $O(b^d)$

* Depth first search (DFS):

Is it Complete? No, unless we delete repeated state

Is it Optimal? Not optimal.

Big O:

- Time complexity: $O(b^d)$

- space complexity: $O(b \times d)$

* Uniform Cost Search (UCS):

Is it complete? yes

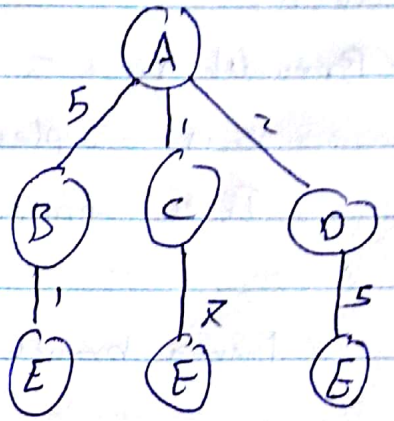
Is it optimal? yes

not well qualified { Time complexity: $O(b^d)$
Space complexity: $O(b^{C/E})$

take: $O(b^{C/E})$ C: shortest path

E: least edge cost

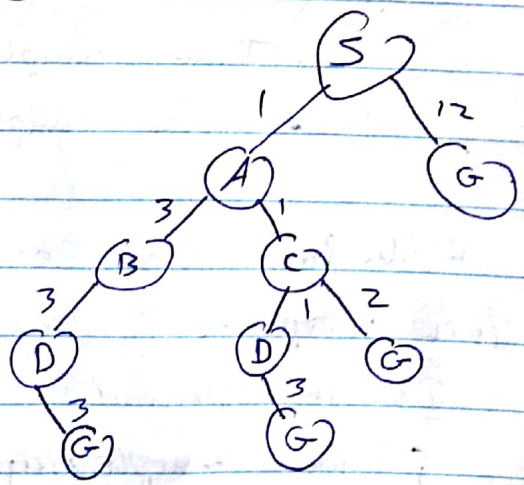
Example:



Expanded Node	Priority Queue
A	A, 0
C	(A → C, 1)
	A → D, 2
	A → B, 5
D	A → D, 2
	A → B, 5
	A → C → E, 8
B	A → B, 5
	A → D → E, 7
	A → C → E, 8
✓ E	A → B → E, 6
	A → D → E, 7
	A → C → E, 8

Example:

Exp. Node	Q
S	S, 0
A	S → A, 1
	S → G, 12
C	S → A → C, 2
	S → A → B, 4
	S → G, 12

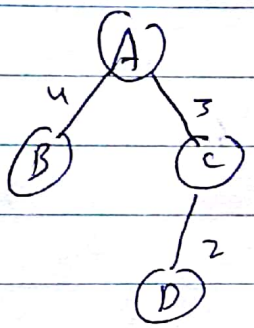


N	Q
D	S → A → C → D, 3 S → A → B, 4 S → A → C → G, 4 S → G, 12
B	S → A → B, 4 S → A → C → G, 4 S → A → C → D → G, 6 S → G, 12
✓ G	S → A → C → G, 4 S → A → C B → D → G, 6 S → A → B → D, 8 S → G, 12

* Heuristic function:

* for water Jug problem
$h(c) = |x - 2|$ (one of the heuristics)

Best First Search:
depends on heuristic value.



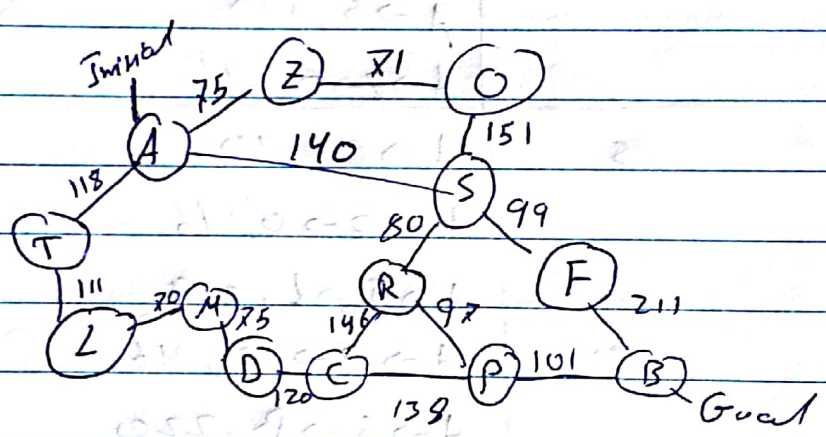
2 lists:
open [B, 4, D, 2
close [A, C, 3

* Function for Best First search: (pseudo code)
open : [Initial state]
close [ ]

While open ≠ empty
- choose best node in open (least heuristic called N
- add N to close
- remove N from open
- if (N is goal) return N
- else generate its successors (depend on rules)

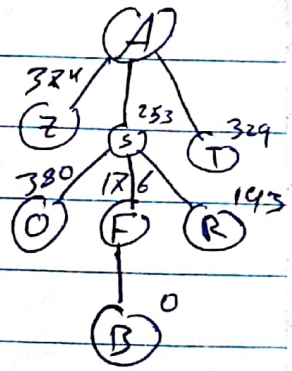
- for each successor in V do //nested loop
  - if the successor was not generated before
    - [evaluate, add it to open, record its parent]
  - else update its parent if the new path is better, update its cost (heuristics)
- return fail.

Example:



Node	hCns
A	366
B	0
C	160
D	242
F	176
L	244
M	241
O	380
P	100
R	193
S	253
T	329
Z	374

Open	Close
A	—
A→S, 253	A
A→T, 329	~~A→S, 253~~
A→Z, 374	~~A→T, 329~~
A→S→F, 176	~~A→Z, 374~~
R, 193	A, S
T, 329	
Z, 374	
O, 380	
B, 0	A, S, F
R, 193	
T, 329	
Z, 374, O, 380	



Path: A→S→F→B

Cost 450

better
Cost: 418
A→S→P→B

Example: Solve the previous example using Uniform cost search:

expanded Node (N)	Priority Queue
A	A, 0
Z	A → Z, 75
	A → T, 118
	A → S, 140
T	A → T, 118
	A → S, 140
	A → Z → O, 146
S	A → S, 140
	A → Z → O, 146
	A → T → L, 229
O	A → Z → O, 146
	A → S → R, 220
	A → T → L, 229
	~~A → S → F, 239~~
R	A → S → R, 220
	A → T → L, 229
	A → S → F, 239
	A → Z → O → S, 297
L	A → T → L, 229
	A → S → F, 239
	A → Z → O → S, 297
	A → S → R → P, 317
	A → S → R → C, 366

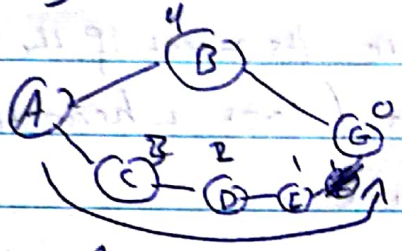
N	Q
F	A → S → F, 239
	A → Z → O → S, 297
	A → S → R → P, 317
	A → S → R → C, 366
	A → T → L → M, 399
S	A → Z → O → S, 297
	A → S → R → P, 317
	A → S → R → C, 366
	A → T → L → M, 399
	A → S → F → B, 450
P	A → S → R → P, 317
	A → S → R → C, 366
	A → T → L → M, 399
	A → Z → O → S → R, 379
	A → Z → O → S → F, 398
	A → S → F → B, 450
C	A → S → R → C, 366
	A → Z → O → S → R, 379
	A → Z → O → S → F, 398
	A → T → L → M, 399
	A → S → R → P → B, 418
R	A → S → F → B, 450
F	⋮
M	⋮
B	A → S → R → P → B, 418

Longer path, but optimal solution

* Best first Search:

Is it optimal? No

Greedy problem



Is it complete? No

↳ Infinite Loop

: Time, space complexity : $O(b^d)$ "Worst case"

A* Search:

function of A* search:

open [Initial state]

close [ ]

While open ≠ empty

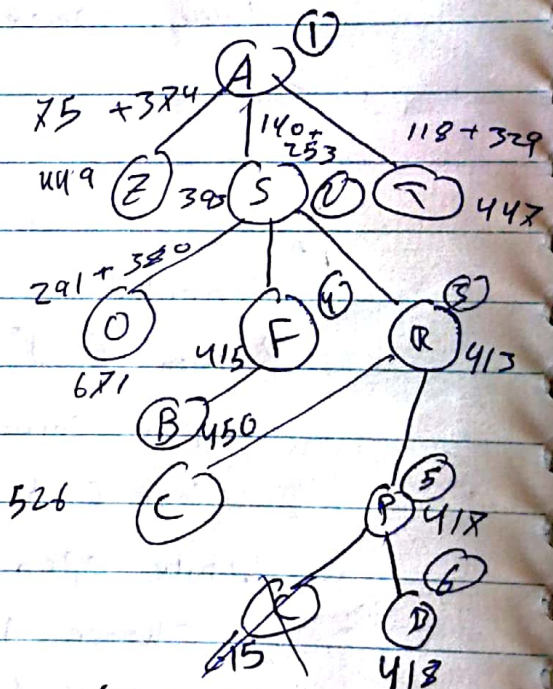
- Choose best node in open (least heuristic + cost)
  called N
- Add N to close
- Remove N from open
- if N is goal, return N
- Else Generate its successors
  (depending on Rules

- for each successor in N do
  - If the successor was not generated before
    → evaluate (cost + heuristic), add to open, record its parent.
  - else, update its parent if the new path is better
    update its (cost + heuristic)
- return fail.

end function.

Example: on previous example (graph on Best first Search)

open	close
A	–
A→Z, 449	A
A→S, 393	
A→T, 447	
A→Z, 449	A, S
A→T, 447	
A→S→O, 671	
A→S→F, 415	
A→S→R, 413	
A→Z, 449	A, S, R
A→T, 447	
A→S→O, 671	
A→S→F, 415	
A→S→R→C, 526	
A→S→R→P, 417	



open	closed
Z, 449	A, S, R, F
T, 447	
O, 621	
~~F, 415~~	
C, 526	
P, 417	
B, 450	
Z, 449	A, S, R, F, P
T, 447	
O, 621	
C, 526	
B, 418	

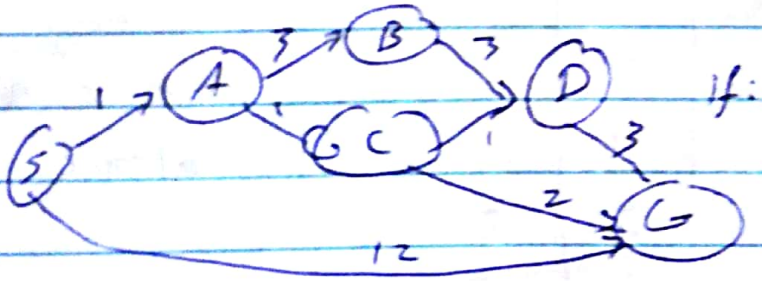
Path: A → S → R, P, B

cost: 419

Tested Nodes: A → S → R → F → P → B

A*:

* Is it complete? yes
* Is it optimal? yes
  - → if it was admissible? ← to check if heuristic is well-estimated for all nodes
  - → monotonic.



If: $h(n) \leq$ shortest path(n) then it is admissible

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

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

h(S) > distance (n)

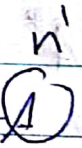
5 > 4

∴ not admissible

* Monotonic:

$$h(n) \leq h(n') - c(n - n')$$

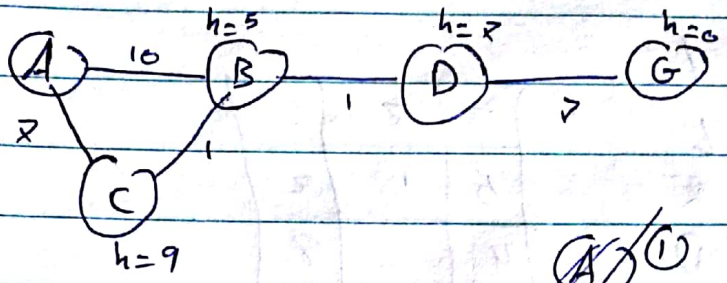
(cost)



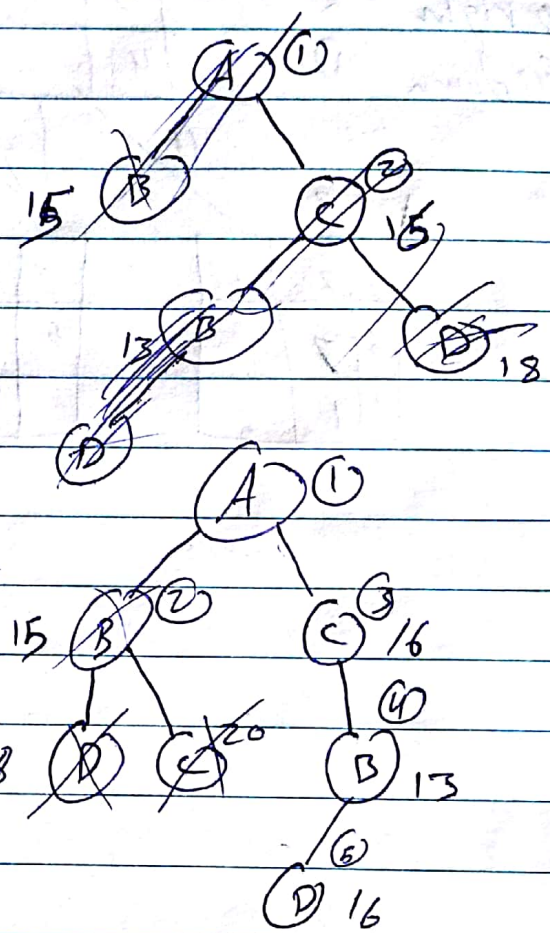
n - n'	h(n)	h(n')	c(n-n')	sum
S - A	5	3	1	4
S - G	5	0	12	12
A - B	3	6	3	9
A - C	3	2	1	3
B - D	6	3	3	6
C - D	2	3	1	4
C - G	2	0	2	2
D - G	3	0	3	3

not monotonic

Example:



open	close
A	-
A → B, 15	A
A → C, 16	
A → B → D, 18	A, B
A → C, 16	
A → C, 16	A, C
A → B → D, 18	
A → C → B, 13	
A → C → B → D, 18	A, C, B
G, 16	A, C, B, D



open	close
A	-
A → B, 15	A
A → C, 16	
A → C, 16	A, B
A → B → D, 18	
A → B → C, 20	
A → C → B, 13	A, C
A → B → D, 18	
A → B → C, 20	

open	close
A → C → B → D, 16	A, C, B
A → B → D, 18	
A → B → C, 20	A, C, B, D
A → C → B → D → G, 16	
A → B → D, 18	G
A → B → C, 20	

Example: solve Eight puzzle using A^*:

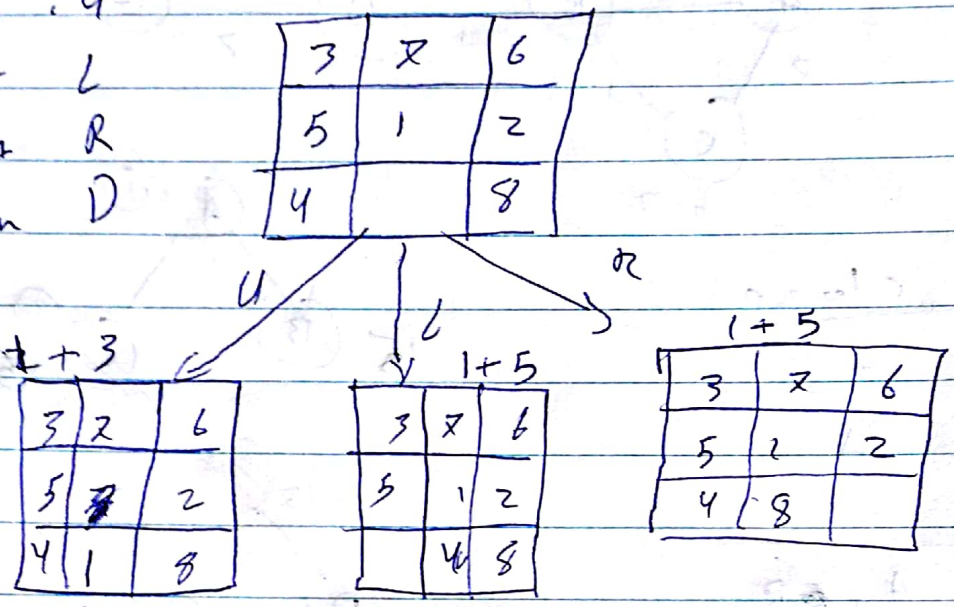
Priority:

① Up : U

② Left L

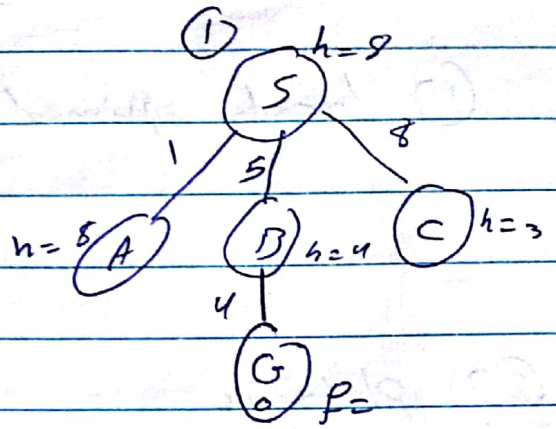
③ right R

④ Down D



Example:

Open	Close
S	-
A, 9	S
B, 9	
C, 11	
A, 9	S, B
G, 9	
C, 11	



* Hill climbing Algorithm:

Current ← random state (initial) from State space

Loop

generate successors (depend on rules)

next ← highest heuristic (successors)

if (h(next) < h(current)) return current

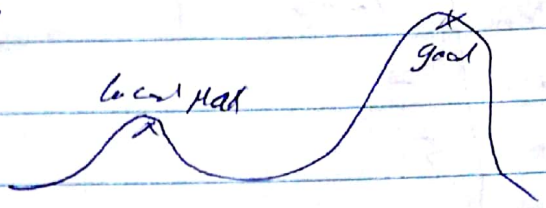
Current ← next

End.

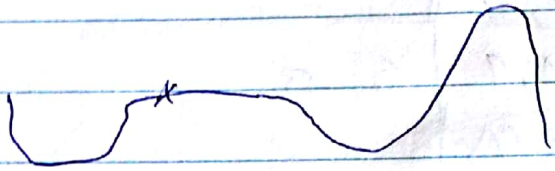
To solve 8-Puzzle with Hill climbing put heuristic to be negative

Draw Back Hill climbing (problems)

(1) Local optimal



(2) plateaus



solution :

Random reset

Time	space	
O(d)	O(b)	(branch factor)

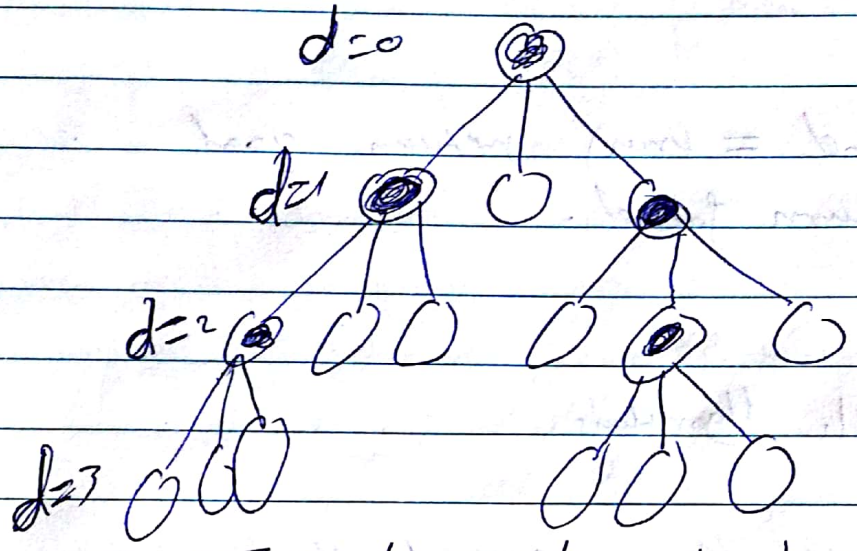
Is it optimal ? no, unless using random reset.

Is it complete ? No, no backtracking

* Beam search :

takes W nodes "Best Nodes"

Suppose b = 3, W = 2



* Expand according to best heuristic

* Space complexity : O(bW)

* Time complexity : O(dW)

* Beam Search Algorithm : two lists: open, W-open

- Found ← false
- generate successor (initial) —add→ open
- while (found = false && open ≠ empty) {
  - choose the best W-nodes from open and add them to W-open (clear open)
  - While (found = false & W-open ≠ empty)
    - get best Node from d-open call it X.

- // inner loop
  - if (x = goal) found ← true
  - else: generate x successors
    add them to open
-
-
- if (found = true) return goal
- else return failed.

// Remark: Search Algorithms:
Blind search

(1) BFS:

complete: Yes, optimal: No, unless cost increase each step equally, → optimal

Space: $O(b^d)$

time: $O(b^d)$

(2) DFS: complete: No, unless we delete repeated node

optimal: No

Time: $O(b^d)$

Space: $O(bd)$

(3) UCS: complete: Yes

optimal: Yes if cost > 0

time: $O(b^{C/E})$ C/E: avg number of nodes that should be visited during search

Space: $O(b^{C/E})$

* Inform Search:

① Best First Search: "Greedy search"

Complete: No, unless delete repeated Node

Optimal: No

Time: $O(b^d)$ } depend on heuristic

Space: $O(b^d)$ }

② A* search:

Complete: Yes

Optimal: Yes, (if h is monotonic)

time: $O(b^d)$

space: $O(b^d)$

③ Hill climbing:

Complete: No

Optimal: No

time: $O(d)$

space: $O(b)$

④ Beam search:

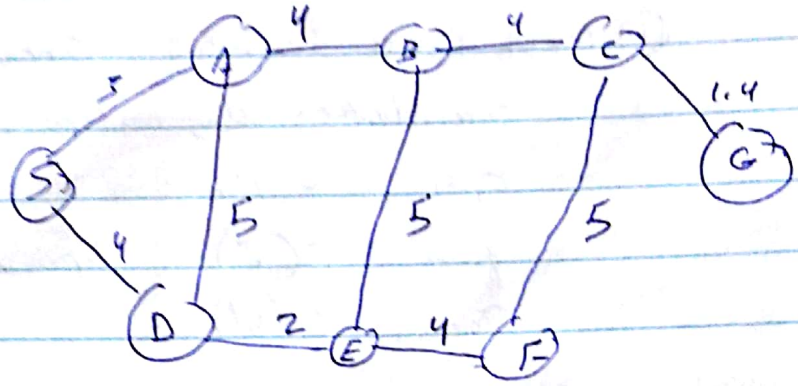
Complete: No

Optimal: No

time: $O(wd)$

space: $O(wb)$

Example :
Exercise 1

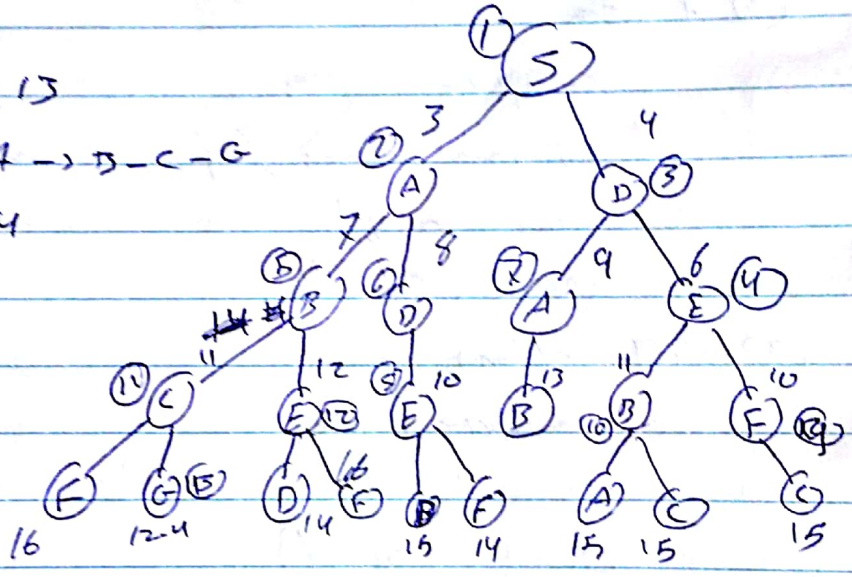


① UCS

Tested Nodes : 15

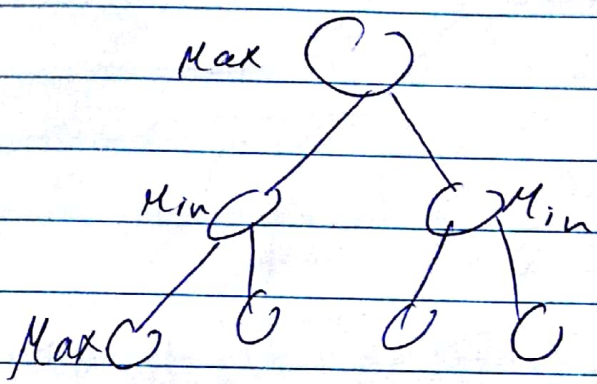
Sol. Path : S → A → B → C → G

12.4



Chapter 3: Game Theory:

Ply: level



Max: computer
Min: player

Algorithm: to determine the next Move

Minimax Theory:

* Two players
* Zero-sum (win, lose or tie)
* perfect
* Deterministic

Algorithm:

```
function minimax (Node, MaxT)        // MaxT: max/min true/false
{ if (terminal)                      // terminal: leaf
      return heuristic (eval. func)  // call func to find eval. value
  if (MaxT) {
      best = -∞
      for each child of Node {
```

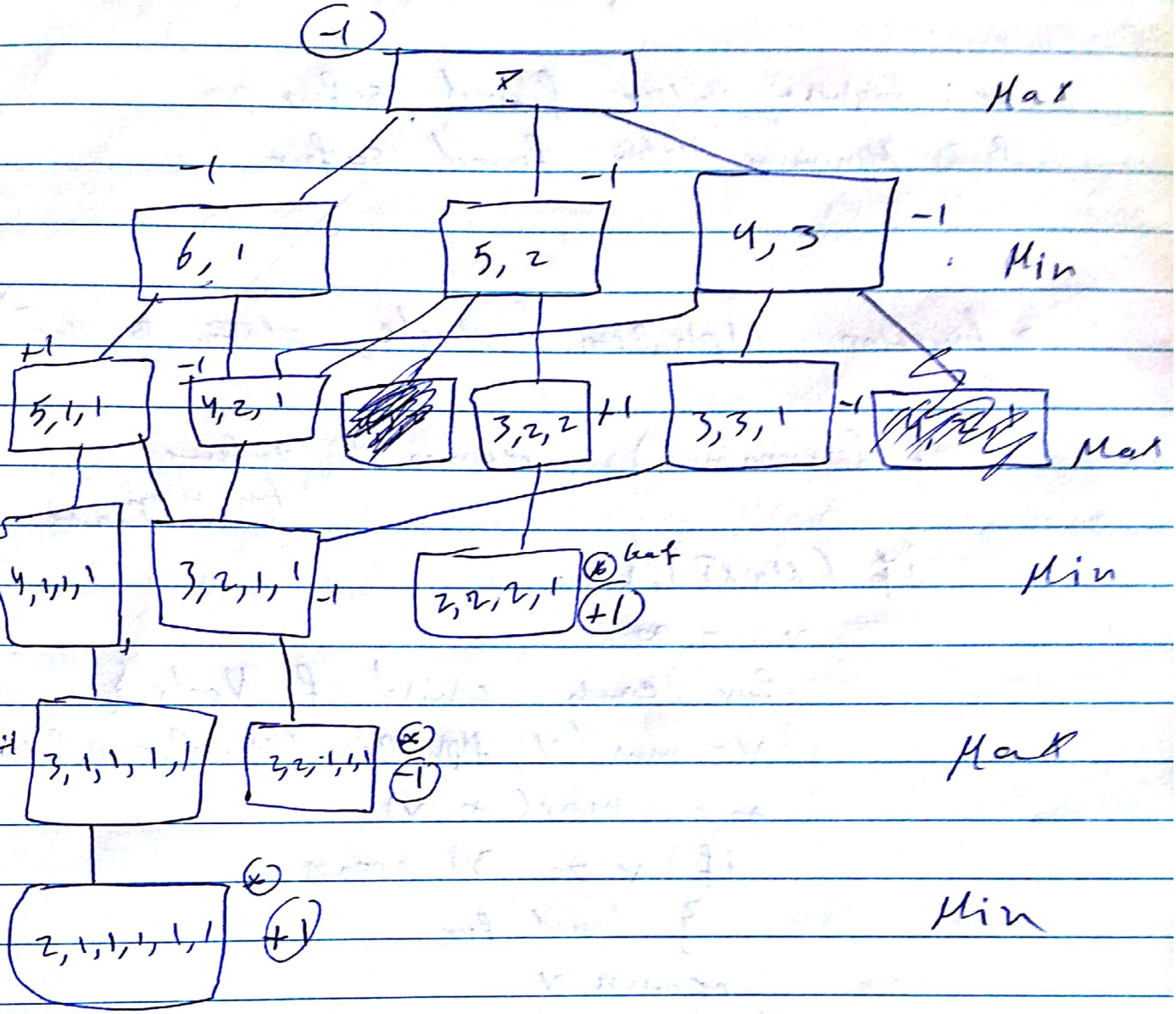
```
        v: minimax(child, false)
        best: max(best, v)
    } // for
    return best
} // if

else {
    best = +∞
    for each child of node {
        v: minimax(child, true)
        best: min(best, v)
    }
    return best
}
```

Time complexity: $O(b^d)$

Space complexity: $O(b \cdot d)$

* Nim Game



$f(n) = 1$ → Win

$f(n) = 0$ → draw

$f(n) = -1$ → Lose

Time complexity: $O(b^d)$

Space complexity: $O(bd)$

* Alpha-Beta Pruning :

α : highest value found so far
β : Minimum value found so far

⇒ function AlphaBeta (Node, ~~alpha~~ α, β, MaxT)
{
if (terminal) return f(v) // value function to evaluate

if (MaxT) {
v := −∞
for each child of Node {
V = max (V, AlphaBeta (child, α, β, false)
α = max (α, v)
if (α >= β) break
} // end for
return v
}
else {
V := +∞
for each child of Node {
V = min (v, AlphaBeta (child, α, β, true)
β = min (V, β)
if (α >= β) break
}
return v
}

Alpha Beta

Time Complexity : $O(b^{3d/4})$ → optimize to $O(b^{d/2})$ if there is cuts

Space Complexity : $O(bd)$

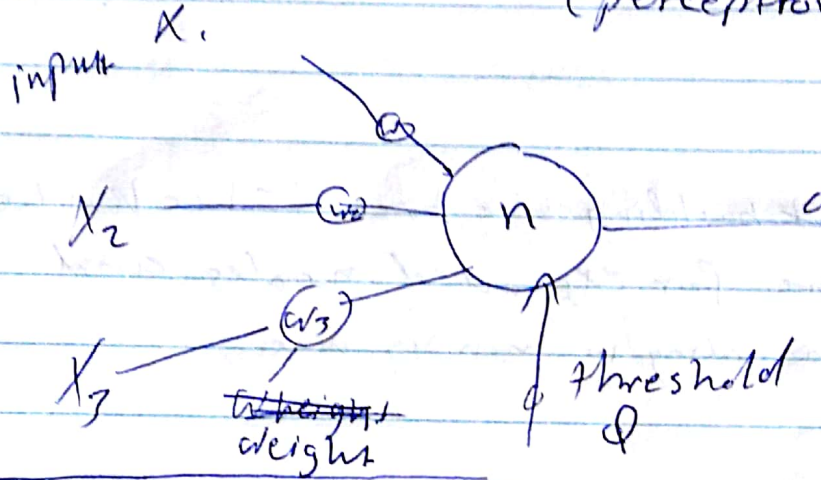
*optimizing α-β:

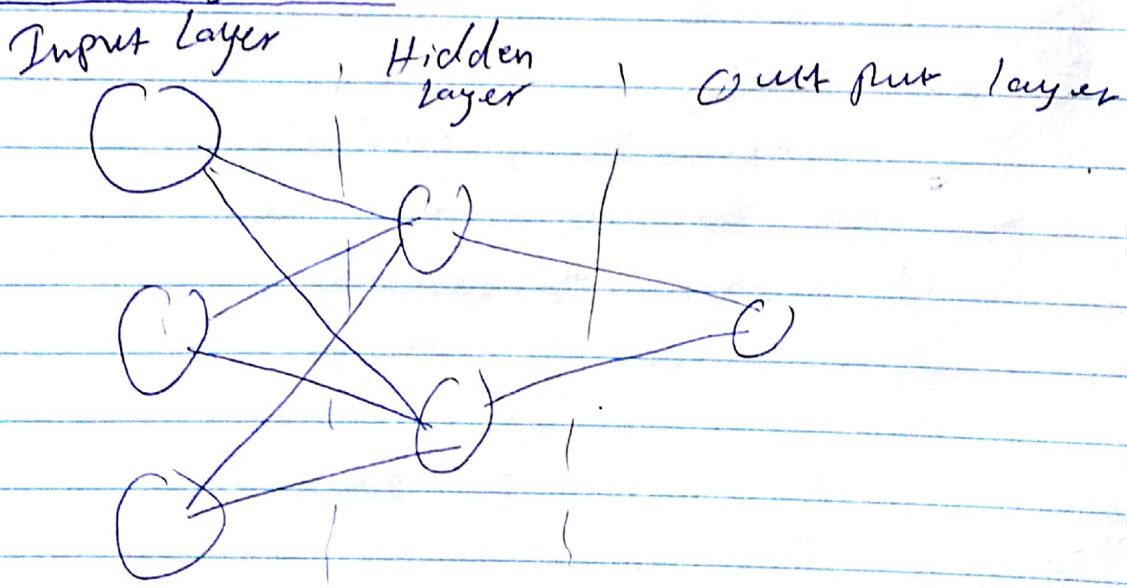
When building the tree, check the estimated value for expanded nodes and sort them according to min or max.

Chapter 4: Machine Learning:

Neural Network:

* Single Node neural network (perception)





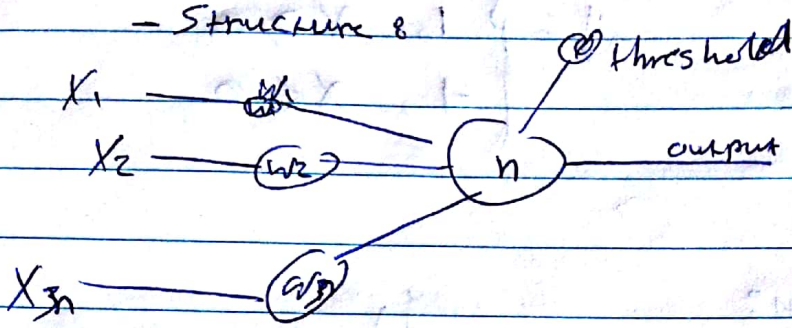
↑ Multi Layer Neural Network.

* all neurons are connected by weighted links

* each neuron receive multi input, produce 1 output

* Single Node Neural Network:

- Structure:



Perceptron: Single node neural network that takes one or more input and produce one output.

* How Perceptron computes its output.

1- Computed weighted sum inputs.

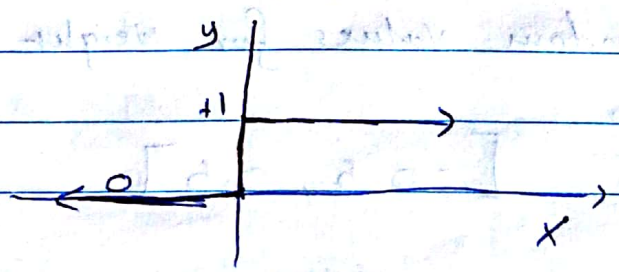
$$X = \sum_{i=1}^{n} x_i w_i - \Theta \quad , \quad \Theta : \text{threshold}$$

n: number of input.

2- apply the weigh sum value to the activation function.

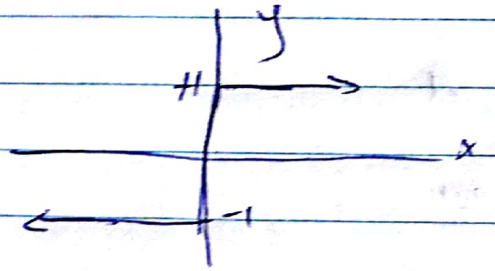
* Activation functions:

(1) step function:



$$y = \begin{cases} 1, & X \geq 0 \\ 0, & X < 0 \end{cases}$$

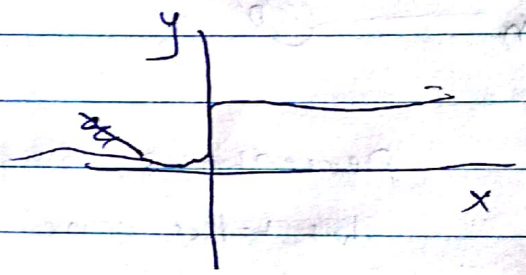
② Sgn function:



$$y = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

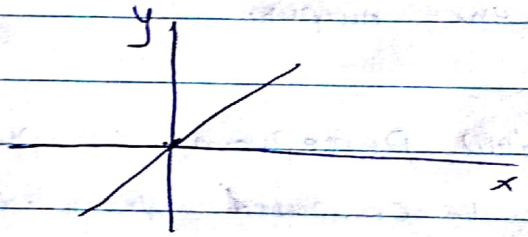
③ Sigmoid function:

$$y = \frac{1}{1+e^{-x}}$$



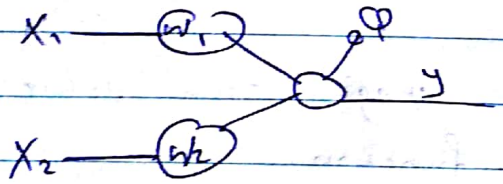
④ Linear function:

$y = x$



* How does perceptron learn:

Keep adjusting the weight until the difference between the actual output and desired output is zero.



* We can do that by doing the following:

① Initialization:

Give random values for weight and threshold.

$[-0.5, 0.5]$

② Activation:

Activate the perceptron by applying the input data (expected)

$X_1, X_2, \dots, X_n \rightarrow$ desired output.

then compute the actual output.

$$y(p) = \text{Step}\left[\sum_{i=1}^{n} x_i(p) w_i(p) - \theta\right]$$

p: this iteration.

then calculate the error

$e = y_{desired} - y_{actual}$

③ Weight training:

calculate the weight correction

$\Delta w_i(p) = \alpha * x_i(p) * e$

α: Learning rate

α: Its a training parameter, it controls the size of the weight (determines the smoothness of the change).

$w_i(p+1) = \Delta w_i(p) + w_i(p)$

④ Iteration (repeat):

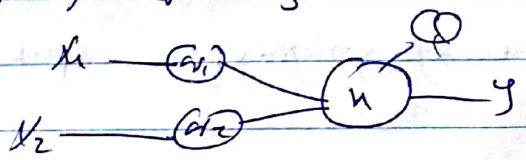
Increase the iteration by 1, then go to step 2 until the error for all iterations in the epoch is Zero.

* Epoch: Single pass through the entire Learning set

iteration	X_1	X_2	X_3	y_d
	0	0	0	1
	0	1	1	0
	0			

→ Epoch

Ex: Perceptron for And Gate:



① Initialization:

from question: $w_1 = 0.3$, $w_2 = -0.1$, $\theta = 0.2$, $\alpha = 0.1$

② Activation:

x_1	x_2	yd	y_a	e
0	0	0	0	0
0	1	0	0	0
1	0	0	1	-1
1	1	1		

i_1: $y_a = step[x_1w_1 + x_2w_2 - \theta]$

$= y[-0.2] = 0$

$e = 0 - 0 = 0$

i_2: $y_a = step[0 + 1 \times 0.1 - 0.2] = step[-0.1]$

$= 0$

$e = 0$

i_3: $y_a = step[0.3 + 0 - 0.2] = step[0.1] = 1$

$e = 0 - 1 = -1$

$\Delta w_1 = \alpha \times x_1 \times e = 0.1 \times 1 \times -1 = -0.1$

$w_{1\,new} = 0.3 + -0.1 = 0.2$

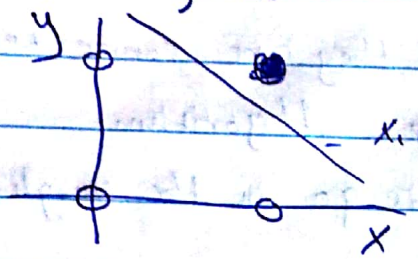
---

* Can Perceptron be trained to recognize any function?

The perceptron can represent only linear seperable functions

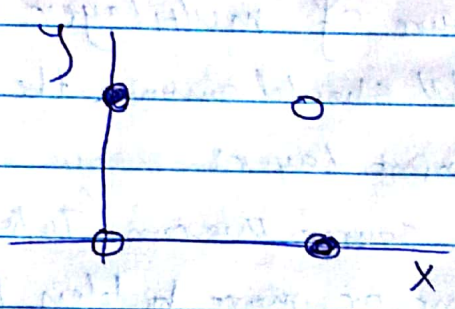
* if there is a line or a plane that can separate the data set into classes, then the perceptron can be trained to recognize that function.

e.g: And gate : ●: 1 , ○: zero



(separable)

e.g: XOR :

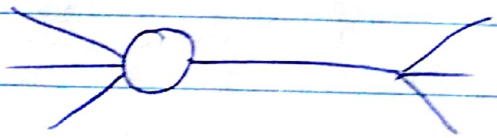


Not separable, can't learn by perceptron.

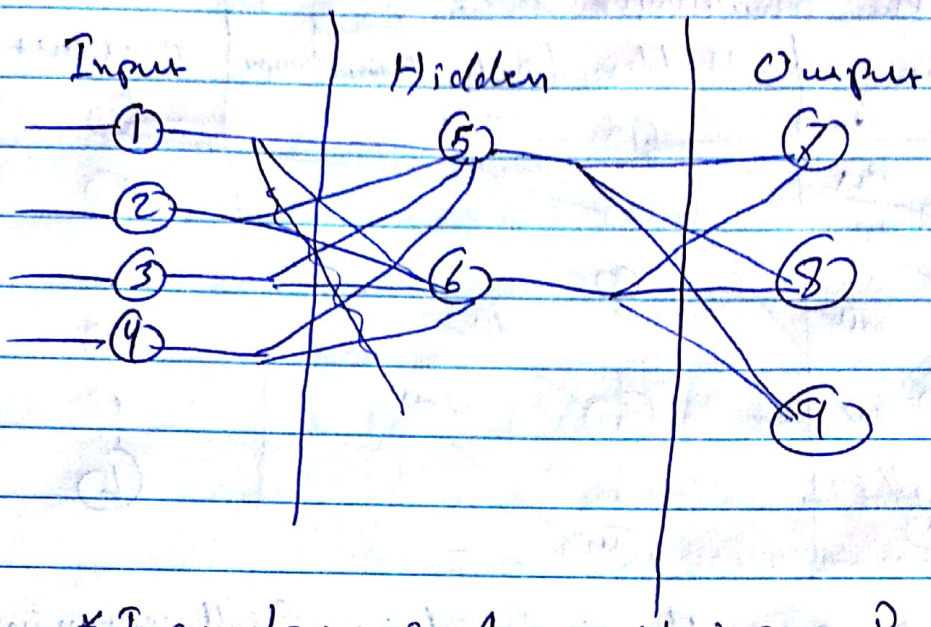
* Multilayer Neural Network:
"Feed forward Neural Network"

* To build multilayer NN:
Structure ← - We need to decide how many neurals and how they are connected.
- Learning Algorithm.
- finally, we go in the weight training

* Structure of multilayer NN:
The NN should contain the following layers:
- Input Layer:
Source Neuron (Taken from environment).
- One or more hidden Layers:
Computation neurons
* Commercial NN has 1-2 hidden layers
* Experiment NN: 3-4 hidden layers
- Output Layer:
Computation Neurons
* Each neuron is connected by weighted link.
weight: Strength of neuron input.
* Each neuron can receive one or more input but produce only one output.

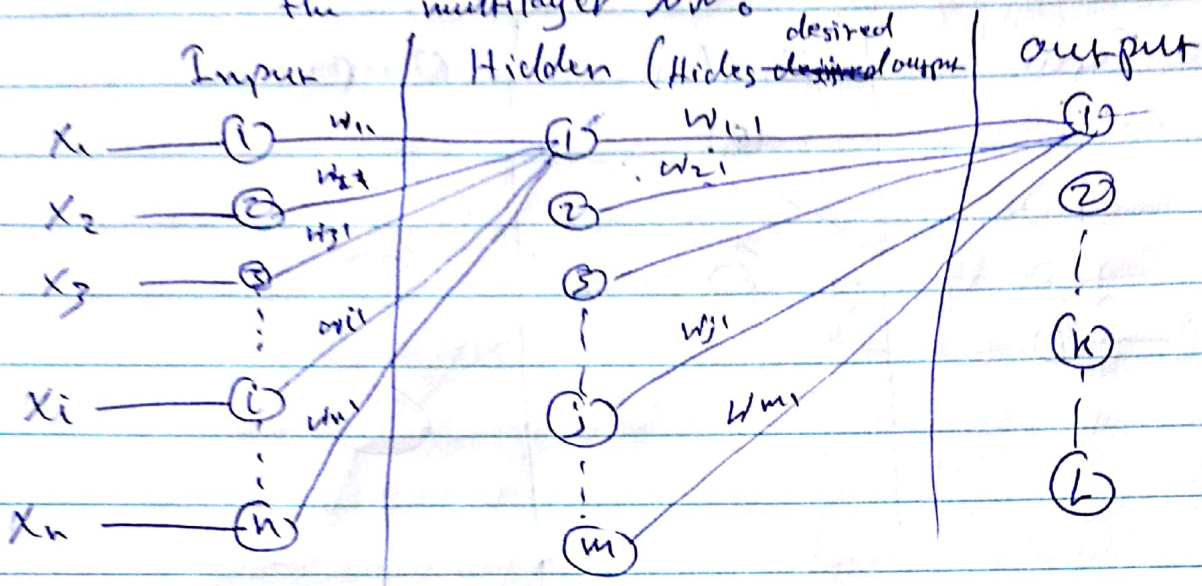


Example:



* Input Layer: Accept the input Pattern from the environment and redistribute it to the hidden layer.
* Hidden layer: Detect the feature hidden in the input layer.
* output layer: Accept the output Pattern from the hidden layer, and compute the final output of the network.

* How to determine the output for the neurons in the multilayer NN:



1- Compute the weighted sum for all neuron input.

2- Apply it to the activation function.

Weighted sum: $X_{H_1} = x_1 w_{11} + x_2 w_{21} + \cdots + x_i w_{i1} + x_n w_{n1} - \theta_{H_1}$

$y_{H_1} = \text{Activ.}[X]$

* Each layer in the same NN may have activation function different from other layers but neurons in the same layer have the same function.

$X_{out_1} = y_{h_1} w_{11} + y_{h_2} w_{21} + \cdots + y_{h_j} w_{j1} + y_{h_m} w_{m_1} - \theta_{out_1}$

$y_{out} = \text{Activ}[X_{out_1}]$

* Learning in multilayer Neural Network:

- Back propagation learning algorithm:

① First, an input pattern is presented to the network.

② The network then propagate the input pattern Layer-Layer until the output is generated.

- if this pattern is different from the desired one, then an error is calculated and propagated backward from the output layer to the input layer.

- Weights are modified while moving from the output layer to the input layer.

* Equations:

① Initialization:

Random Values for weight & threshold.

$$\left[\frac{-2.4}{fi}, \frac{2.4}{fi}\right]$$

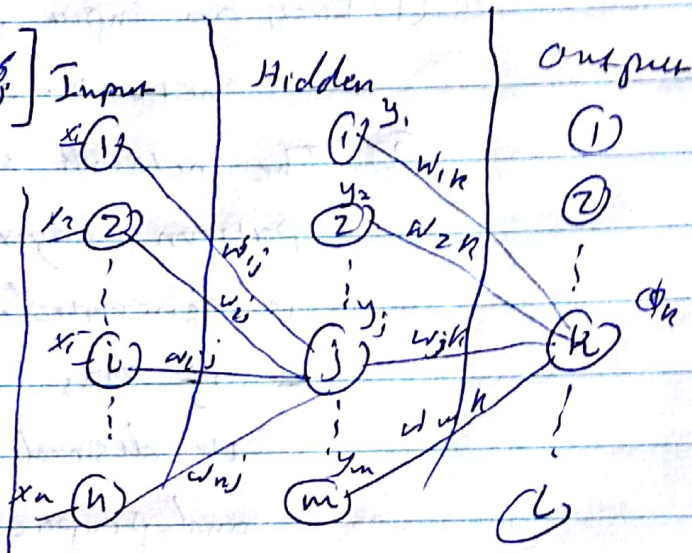
fi: Number of input for neuron i

② Activation: "forward"

- Activate multilayer neural network by using the input $X_1, X_2, \ldots X_n$ & output $Yd_1, Yd_2, \ldots, Yd_n$.

a- Compute the actual output for all neurons in the hidden Layer.

$$y_j = \text{Sigmoid}\left[\sum_{i=1}^{n} x_i w_{ij} - \theta_j\right]$$



b- Compute the actual output for all neurons in the output Layer.

$$y_{k_{\text{actual}}} = \text{Sigmoid}\left[\sum_{j=1}^{m} y_j w_{jk} - \theta_k\right]$$

(3) Weight training & "Part of learning, weight correction"

- calculate error $(e_k) = y_{dk} - y_{ak}$

@ Compute error gradient (δ_k) for all neurons in the output Layer.

$$\delta_k = y_{a_k} * [1 - y_{ak}] * e_k$$

$$\Delta w_{jk} = \alpha * y_j * \delta_k$$

$$w_{jk_{new}} = \Delta w_{jk} + w_{jk\,old}.$$

(b) Compute the error gradient for all neurons in the hidden Layer.

$$\delta_j = y_j * [1 - y_j] * \sum_{k=1}^{l} w_{jk} * \delta_k$$

$$\Delta w_{ij} = \alpha * x_i * \delta_j$$

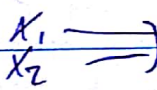
$$w_{ij,new} = \Delta w_{ij} + w_{ij,old}$$

(4) Repeat :

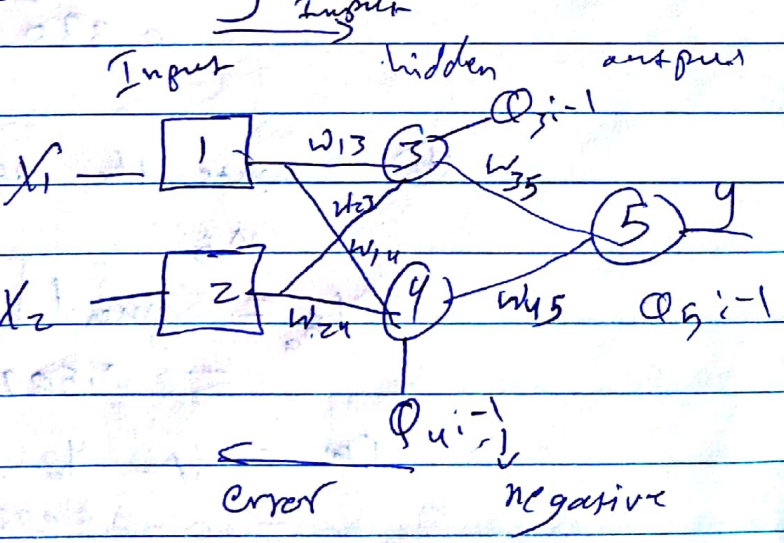
- Go to Step 2
- SSE : Sum Square error $\sum_{i=1}^{n} e_i^2 \simeq 0.001$ iteration
- MSE : Mean square error $\frac{1}{n} \sum_{i=1}^{n} e_i^2 \simeq 0.001$

---

* Multilayer NN for solving XOR :



x_1	x_2	y_d
0	0	0
0	1	1
1	0	1
1	1	0



(1) Initialization:

$W_{13} = 0.5$, $W_{35} = -1.2$ $\theta_3 = 0.8$

$W_{23} = 0.4$, $W_{45} = 1.1$ $\theta_4 = -0.1$

$W_{14} = 0.9$ $\theta_5 = 0.3$

$W_{24} = 1.0$

(2) Activation:

Assume we apply the following training set input $x_1 = x_2 = 1$, $y_d = 0$

(a) Compute the actual output for n_3, n_4

$$y_3 = \text{Sigmoid}\left[W_{13} \cdot x_1 + x_2 W_{23} - \theta_3\right]$$

$$= \text{Sigmoid}\left[0.5 + 0.4 - 0.8\right]$$

$$= \frac{1}{1 + e^{-0.1}} = 0.525$$

$$y_4 = \text{Sigmoid}\left[x_1 W_{14} + x_2 W_{24} - \theta_4\right]$$

$$= 0.8808$$

(b) Compute the actual output for neuron n_5

$$y_{5_a} = \text{Sigmoid}\left[y_3 W_{35} + y_4 W_{45} - \theta_5\right]$$

$$= 0.5097$$

$$e = y_d - y_a$$

$$= 0 - 0.5097 = -0.5097$$

(3) Weight Training :

$$\delta_5 = y_{a5}(1 - y_5) * e$$
$$= 0.5097 (1 - 0.5097) * -0.5097$$
$$= -0.1274$$

Weight correction

$$\Delta W_{35} = \alpha * y_{a3} * \delta_5$$
$$= -0.0067$$

$\alpha = 0.1$

$$\Delta W_{45} = \alpha * y_{a4} * \delta_5$$
$$= -0.0112$$

$$\Delta \theta_5 = \alpha (-1) * \delta_5$$
$$= 0.0127$$

old (befor correction)

$$\delta_3 = y_3 (1 - y_3) * W_{35} * \delta_5$$
$$= 0.0381$$

$$\delta_4 = y_4 * (1 - y_4) * W_{45} * \delta_5)$$
$$= -0.0147$$

$$\Delta W_{13} = \alpha * x_1 * \delta_3$$
$$= 0.0038$$

$$\Delta W_{23} = \alpha * x_2 * \delta_3$$
$$= 0.0038$$

$$\Delta W_{14} = \alpha * x_1 * \delta_4$$
$$= -0.0015$$

$$\Delta W_{24} = \alpha * x_2 * \delta_4$$
$$= -0.0015$$

$$\Delta \theta_3 = \alpha * -1 * \delta_3$$
$$= -0.0038$$

$$\Delta \theta_4 = \alpha * -1 * \delta_4$$
$$= 0.0015$$

New Weights:

$W_{13} = 0.5038$	$W_{35} = -1.2067$
$W_{14} = 0.8985$	$W_{45} = 1.0888$
$W_{23} = 0.4038$	$\theta_3 = 0.7962$
$W_{24} = 0.9985$	$\theta_4 = -0.0985$
	$\theta_5 = 0.3127$

---

* Accelerating Learning:

$\Delta W_{jk}(p) = \alpha \cdot y_j \cdot \delta_k$; with momentum

(p ← iteration)

With momentum: $\beta = 0 \le \beta \le 1$

$\Delta W_{jk}(p) = \beta \Delta W_{jk}(p-1) + \alpha \cdot y_j \cdot \delta_k$

---

* Remark:

* Supervised Neural Network: (there is Input, output)
  - Feed forward N.N
    - Pattern recognition
    - function fitting
    - prediction
  - Radial Basis N.N.

* Unsupervised Neural Networks:

"You don't know the output"

- They found the relationship within the data.

Eg: Self organizing Map (SOM)

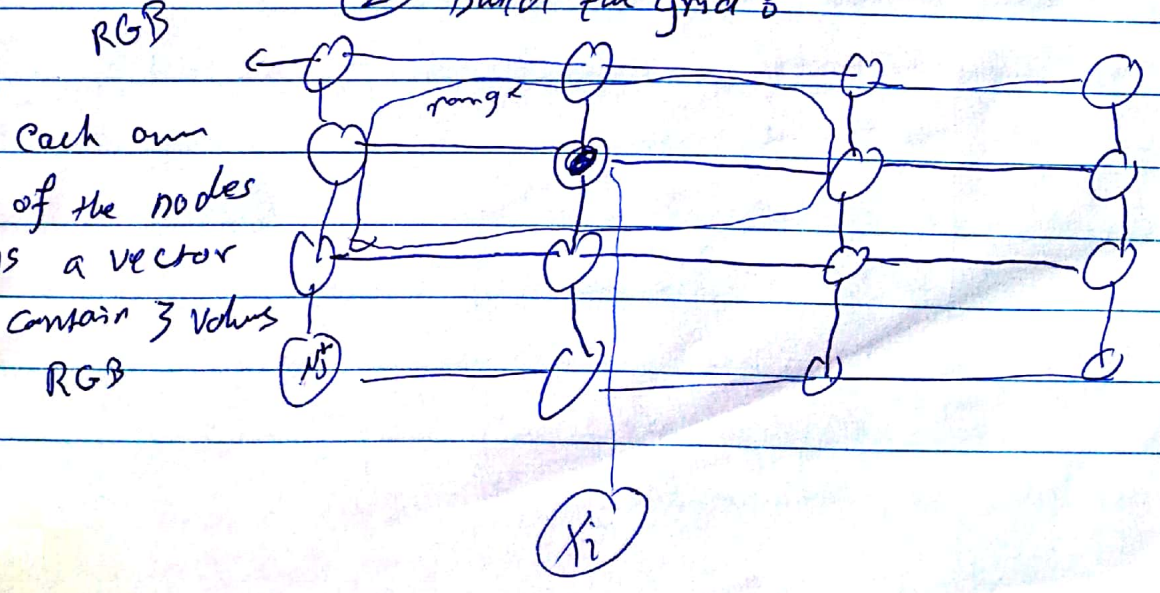
- learn to classify input vector regarding to similarity.

* How self organizing Map learn?

① Bring the input data.

	R	G	B	
X =	0	0	0	→ black
	0	0	1	→ blue
	0	1	0	⋮
	0	1	1	
	1	0	0	
	1	0	1	
	1	1	0	
	1	1	1	→ white

② Build the grid:



(3) Initialize each node with random color.

(4) Choose an Input and then compare it with all the nodes in the grid
- Select node with minimum distance "best matching unit" (BMU)

$$d_{ij} = \sqrt{\sum_{k=1}^{m} (x_{ik} - w_{jk})^2}$$

m: number of values in Node (3)

(5) Update for each Node in the range of BMU

$$w_j' = w_j + \alpha W_j (x_i - w_j)$$

α: learning rate

W_j: Weight parameters depend on distance

$W_j = e^{-\frac{d^2}{2r^2}}$; r: radius, d: distance

(6) Repeat the process with different inputs

(7) Repeat with smaller radius (of the range).

Chapter 4: Expert Systems:

* Perform at human level in a narrow domain
* Use ~~static~~ symbolic reasoning
  - Rules, facts (knowledge)
* Apply heuristic to guide through the process.
  - Reduce time and space.
* Provide explanation facility
* May do mistakes.

---

* To develop an expert system:
  there are 5 members:

1- domain expert: Skilled person capable of solving problem in a specific area. He has the greatest expertise in a given domain.

2- Knowledge Engineer:
  Someone who's capable of designing, building and testing an expert system
  - He interview the domain expert to find how a particular problem is solved.
  - He need to select the reasoning method (regarding to the expert)
  - He need to select a development software or an expert shell to represent the knowledge taken from an expert in a language that the machine can understand

3- Programmer: if there is no software or shell then we need a programmer to encode the information in a language that machine can understand.

- He needs AI languages s.a; LISP, Prolog + C, C#, Java

4- Project Manager:

The Leader of the expert system development team. Responsible for keeping the project on track.

5- End User:

The person that uses the expert system

---

* Structure of the expert System:

was named: "production system model".

- based on the idea that human solve problem by using their knowledge.

data { - long term memory
- short term memory

- reasoning
- conclusion

* Basic Components of expert systems:

1- Knowledge Base (KB): fixed

Rules → relations → recommendations
Rules → strategy
Rules → heuristic

2- DataBase of Facts (DBF): Current situations
fluid, differs by different users.
- the rules enable expert system to derive new facts.

3- Inference engine:
KB + DBF —Process→ conclusions

- Forward chaining:
Data —Process→ Decision

- Backward Chaining:
Decision —Process→ Data, then check yes or No

4- Explanation Facility.

5- User Interface: the only thing that differs from expert system to another.

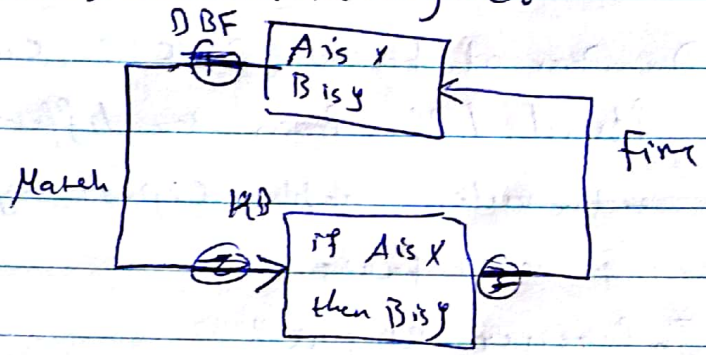
6- Developer interface:
Text editor — to modify the rule.

7- External interfaces:
- External Program: back keeping facility
- External database: runtime knowledge.

* Inference Engine Algorithm:

- Compares each rule with facts it already knows
- When the "if" part of the rule match the facts, the rule will be fired (then part of the rule will be added to the DBF

* Match-Fire cycle:



* The matching of the rule "if part" to the facts produces an inference chain.

* The inference chain indicates how Expert System applies the rules to reach the solution.

Example: "Forward Chaining":

KB:

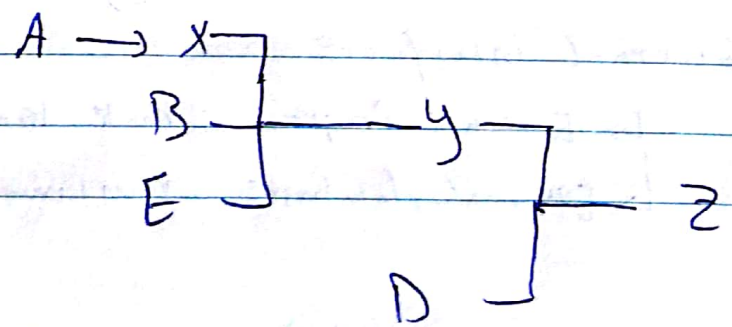
① if y & D then Z

② if X & B & E then y

③ if A then X

DBF | A, B, C, D, E

added | X
y
Z → goal



* The inference engine must decide when the rules be fired.

* There are two ways in which rules are executed:
  - forward chaining
  - Backward chaining.

---

* Forward Chaining:
  - applies rules with known data to achieve the desired goal.
  - called data-driven reasoning.
  - each iteration, topmost rule is executed
  - When fired, a rule added a fact to DBF.
  - The match-fire cycles stops when the goal is reached or no more rules can be fired.

Example: KB: ① $y \& D \rightarrow z$, ② $x \& B \& E \rightarrow y$
③ $A \rightarrow X$ ④ $c \rightarrow L$ ⑤ $L \& m \rightarrow N$

Added: X, L, y, z

DBF | A, B, C, D, E |

cycle 1: Rule 3 fired, fact X is added to DBF
Rule 4 fired, fact L added
cycle 2: Rule 2 fired, fact y added
cycle 3: Rule 1 fired, fact z added

Stop

Example: KB: ① A → x ② Q → T, ③ A → y ④ A ∧ D → Φ
⑤ Q → R, ⑥ R → S ⑦ N → L, ~~T → Z~~
⑧ T → Z

DBF [A, B, C, D, E] goal is Z

Added X, Y, Q, ~~T~~, R, S, T, Z

cycle 1: rule 1 fired, fact X added
rule 3 fired, fact y added
rule 4 fired, ~~fact~~ Q added
~~rule 2 is fired fact T added~~
~~cycle 2: rule 2 fired, fact T added~~
~~cycle 2~~ rule 5 fired, fact R added
~~cycle 3:~~ rule 6 fired fact S added
cycle 4: rule 8 fired, fact Z added
~~Stop~~

Ex: in previous example, if the goal was L

We need 3 ~~4~~ cycles → don't reach the goal
→ time consuming.

* Using forward chaining, many rules maybe executed unnecessarily.

* Backward Chaining :

- In BC, the expert system has the goal, and the inference engine attempt to find the evidence to prove it.
- First the knowledge base is searched to find the rules might have desired solution such rule must have the goal in their "Then part".
- If such rule is found and its "if part" match data in the database, then the rule is fired and the goal is proven.
  However, this is rarely the case
- If not, the IE (inference engine) Stack the rule and set up a new goal (subgoal)
- then the KB is searched again for rules that can prove the subgoal

---

Example: slide55: D B F | A B C D E

* goal is Z

Pass 1:
  Stack rule (1) (Y & D → Z)
  setup goal Y → subgoal.

Pass 2: Stack rule (2) (X & B & E) → Y
  setup new goal X, subgoal

~~Pass 3~~

Pass 3: fire X

~~Example : depend on Example 2 of FC~~

* Conflict Resolution :

assume we have the following :

Rule 1 : if the traffic light is green then action is go.

Rule 2 : if traffic light is red then action is stop.

Rule 3 : if the traffic light is red then action is go.

"More than one rules with same if part but different then part".

---

* Conflict resolution :

Rules with identical if part can cause conflict via their then parts.

* The inference engine must determine which rule to fire.

* The method for choosing rule to fire when more than one rule can be fired in a given cycle is called conflict resolution

*Methods used for conflict Resolution:

1- Highest priority:

In simple ~~example~~ application the priority can be established by placing the rules in appropriate order.

2- Longest Matching ~~Rule~~ Strategy:

fire the most specific rule

- if the season is autumn and the sky is cloudy and the forecast is rain then the advice stay home
- if season is autumn then the advice is ~~is~~ take an umbrella

* applies the first rule since it has more in "if part".

3- data most recently entered in the database:

- this method relies on time tag attached to each fact in the database

11/10/2017

- If the forecast is rain → 8:16 pm
  then the advice is take umbrella
- If the weather is wet → 10:18 12/10/2017
  then advice: stay home.

⇒ rule 2 is fired.

Example: Longest match:

if A & B & C then D is true

if A then D is false

Example: most recent updated:

if B [8:00 2/9/2017]

then D is true

if A [9:00 3/9/2018]

then D is false.

---

* Meta Knowledge:

Knowledge about Knowledge

tells what rules has more priority

Expert system vs expert:

Expert System:
- consistency in expert system.
- Memory is better in expert system.
- Access for expert system anytime.
- Availibity for many users at a time
- Never dies

* disadvantages :- (expert system)

Predicted

No emotions.

Dump ; No common sense

Does not learn by itself.

Chapter 5: Prolog:

* facts: - Should always begin with small letter and end with full stop.
  - Then it can contain any letter or number
  - No space → use underscore instead
  - No symbols (-, +, * ...)

Ex: D.B, Place to write code

```
sun.
today_Is_sunday.
ali_Eat_apple.
```

Query, (Testing, Matching)

```
?- sun.
Yes
?- today_Is.
No
?- raining.
No
```

Ex: Which of the following are correct facts:

✓ - tomsRed Car.

X - 2 Ideas.

X - Prolog

Ex:

```
blue_box.
red_box.
green_circle.
blue_circle.
Orange_triangle.
```

```
?- green_circle.
true
?- circle_green.
fals
?- red_triangle.
false.
```

* facts With arguements :

* more complicated facts, consists of relations (predicate) and items (parameters).

predicate ← relation (arg1, arg2, ...).

Start small letter

e.g : likes (emad, football).
eats (emad, apple).
eats (ali, orange).
eats (ali, steak).

query :

?- eats (emad, apple).
true

?- eats (apple, emad).
false.

Ex: age (ali, 20).
age (farah, 3).

?- age (ali, 20)
Yes

?- age (farah, three).
No

Ex: eats (ali, apple).
eats (farah, steak).

unify (try to match item with variable).

?- eats (Ali, What).
No

?- eats (ali, What).
What = apple

variables (capital beginning)

* Variable :

- Start with capital letter.
- The process of matching items with variable is called unification.

Ex: Loves (emad, football).
loves (ali, train).
?- loves (emad, X)
X = football.
?- loves (train, Who)
false

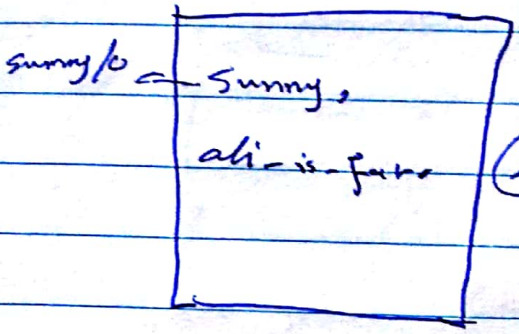
---

visit (emad, jerusalem, uk, nablus).
visit (peter, us, jordan, spain).

?- visit (emad, X, Y, Z).
X = jerusalem
Y = uk
Z = nablus

?- visit (peter, uk, jordan, X).
No

* Simple facts:



Ex: Do these unify?

1- eats(fred, food).
eats(Person, jim).
Yes

2- cd(ra, salt, sugar).
?- cd(A, B, help).
No

3- f(foo, L).
?- f(A1, A1).
✓ L=A1 → f(foo, foo).

4- f(X, a).
?- f(a, X).
Yes

5- likes(jane, X).
?- likes(X, jim).
No

6- f(X, Y)
?- f(P, P)
X=Y → Yes.

* Logical operators:

if → :-

AND → ,

OR → ;

NOT → not

---

* Rules:

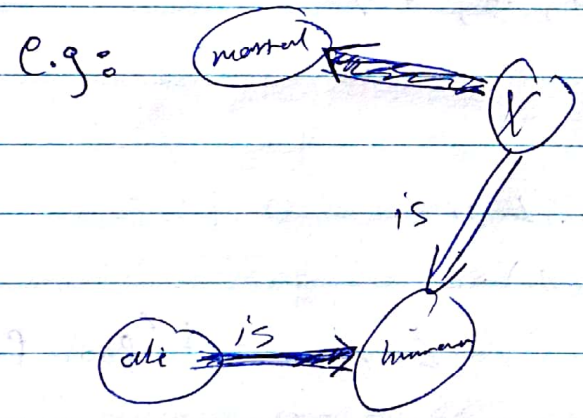
All humans are mortal.

Prolog: conclusion condition

mortal(X) :- human(X).

→ for given X, X is mortal if X is human

e.g.:



KB + DBF

```
mortal(X) :- human(X).
human(ali).
```

?- human(ali). true

?- human(X). X = ali

?- mortal(ali). yes

?- mortal(X). X = ali

Ex:

```
mortal(X) :- human(X).
human(ali).
human(X).
```

?- mortal(farrah). Yes

?- mortal(ali). Yes
Yes

Ex:

```
fun(X) :- red(X), car(X).
fun(X) :- blue(X), bike(X).
car(ford).
car(seat).
bike(bmw).
red(bmw).
red(vw).
blue(bmw).
```

?- fun(bmw). % car(bmw) not found
true. % continue to line 2

Ex: Are these correct syntax.

① a :- b, c, d :- ef. X

② happy(X) :- a, b. ✓

③ happy(X) :- has_money(X) & has_friend(X). X
(& → symbol; must be ,)

④ fun(fish) :- blue(sea), bike(ram). ✓

* Search (Backtracking) :-

Backtracking with facts :

```
eats(ali, apple).
eats(ali, orange).
eats(ali, steak).
```

```
?- eats(ali, What).
What = apple ;        % if put ; searches for another
What = orange .       % if put . it stops.
```

---

* Back Tracking with Rules :

```
hold_party(X) :- birthday(X), happy(X).
birthday(tom).
birthday(fred).
birthday(helen).
happy(mary).
happy(jane).
happy(helen).

?- hold_party(Who).
Who = helen.
```

Exercise:

```
a(X) :- b(X), c(X), d(X).
a(X) :- c(X), d(X).
a(X) :- d(X).
b(1).   ; d(10).
b(a).   ; d(11).
b(2).   { c(3).
b(3)    ( c(4).
```

```
?- a(X).
X = 10
X = 11
```

Ex: if d(3) added,

```
?- a(X).
X = 3
X = 3
X = 3
X = 10
X = 11
```

if b(X), c(X), !, d(X).

```
?- a(X).
false
```

↑

```
?- a(X)
X = 3
```

Ex:

likes(john, mary).
likes(john, trains).
likes(peter, cars).
likes(P1, P2) :- hobby(P1, H), hobby(P2, H).
hobby(john, football).
hobby(tim, sailing).
hobby(helen, football).
hobby(simon, sailing).

added for next question | hobby(john, sailing).

5+

(1) ?- likes(john, train).
Yes

(2) ?- likes(helen, john).
Yes

(3) ?- likes(tim, helen).
No

(4) ?- likes(john, helen).
Yes

(5) ?- like(john, Who).
Who = mary
Who = trains
Who = john
Who = helen
Who = tim
Who = simon
Who = john

* Recursion in Prolog :

Example :

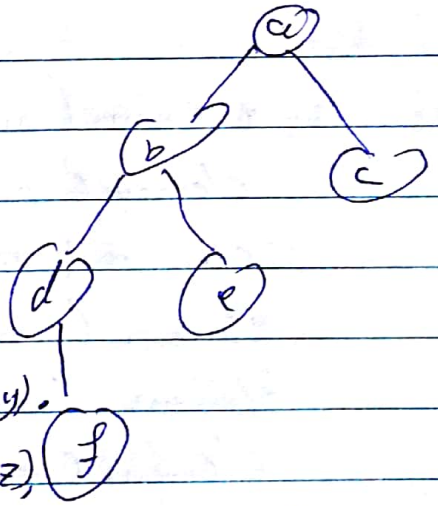
```
on_route(rome).        % stop point.
on_route(Place) :- move(Place, M, Next),
                   on_route(Next).

move(rome, taxi, london).
move(london, train, manchester).
move(manchester, plane, rome).
```

```
?- on_route(rome).
yes.
```

Ex:

```
parent(b, a).
parent(c, a)
parent(d, b)
parent(e, b)
parent(f, d)
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z),
                  ancestor(Z, Y).
```




```
?- ancestor(f, a)
yes.
```

Ex: a is taller than b
b is taller than c
c is taller than d

write recursive program that will determine that [a] height is greater than [d]

taller(a,b).
taller(b,c).
taller(c,d).
h-greater(X,Y) :- taller(X,Y).
h-greater(X,Y) :- taller(X,Z), h-greater(Z,Y).

?- h-greater(a,d)
true

Ex: element(pizza, sauce).
element(pizza, ham)
element(pizza, meat).
element(ham, salt).
element(sauce, tomato).
element(sauce, water).
element(sauce, pepper)
element(sauce, salt).

Write a prolog program [component] such that component(T_1,T_2) succeeds if T_1 is an ingredians in T_2 either direct or indirect.

Solution:

component(T1, T2) :- element(T2, T1).

component(T1, T2) :- element(T, T2), component(T, T1).

?- component(ham, pizza).

yes

?- component(tomato, pizza)

yes.

Other solution:

component(T1, T2) :- element(T2, T1).

component(T1, T2) :- element(T3, T1), element(T2, T3).

?- component(tomato, pizza).

* Arithmetic operators:

+

-

*

/ → real

// → integer

e.g: 97 / 10 = 9.7

97 // 10 = 9.

mod

** → power

e.g. 2 **3 → 8

* in most languages 2+3 is an expression.

* in prolog 2+3 is just a data.

* in prolog = → is used for unify

- it does not evaluate.

e.g: ?- X = 2+3

X = 2+3

?- 2+3 = 3+2

No

?- 5 = 3+2.

No

?- 2+3 = 2+3.

Yes

* To compute arithmetic expression, prolog has builtin predicate called **is/2** (two arguments).

is (item1, item2).
- item1 → variable / Number
- item2 → expression

?- is (X, 2+3).
X = 5.

?- is (5, 2+3).
yes

if not a variable → error

?- is (Y, 2).
Y = 2

?- is (Y, Y+1).
error.

?- Y = 3, is (X, Y+1).
Y = 3
X = 4

error → * second argument in predicate (is) must be grounded (initialized).

false → * first parameter should be a number or variable name (not expression).

* **is** predicate is an infix operator

?- is (X, 2+3).

?- X is 2+3

same as +, -, *, / → 2+3 or +(2,3)

eg: ?- is (X, +(2,3))
X = 5

? X is +(2,3).
X = 5

?- 3+2 is X
Error.

---

* Priority : * / higher priority + -
- left to right.

e.g ?- X is 3+4*5, Y is (3+4)*5
X = 23
Y = 35

?- X is 5-3-1, Y is 5-(3-1).
X = 1
Y = 3

---

* built-in predicate, display1
e.g: ?- display (3+4*5).
+(3, *(4,5))
?- display ((3+4)*5).
*(+(3,4), 5).

?- display (ali).
ali
?- display (raslan kiwan). % error
?- display ('raslan kiwan').
raslan kiwan

Example :- Write prolog program to compute square of a number.

```
p(X) :- Y is X*X, display(Y).
?- p(5).
25
true.
?- p(X)  % error, X not grounded.
```

other sol: `p(X,R) :- R is X*X.`

```
?- p(5,Y).
Y=25
true.
```

Example :- Write a successor predicate that succeed if its second parameter is arithmetic successor of its first.

```
p(X,Y) :- Y is X+1.
?- p(2,3)
true
?- p(2,4)
Y=3
?- p(4,4).
false
```

Example: Write a prolog program to get the last two digit of a given year and print them

```
?- last_two(1985).
85
```

Soln

```
last_two(X):- Y is (X mod 100), display(y).
?- last_two(1985).
85
true
```

---

* Relation Operator:

<	less
=<	less equal
>	greater
>=	greater equal
=:=	equal equal } for numbers only.
=\=	not equal }

```
?- 3 < 5
true
?- 2+3 =< 2+3
true
?- 2+3 = 3+2
false
?- 2+3 =:= 5
true.
```

Example: Write a program to find min of 2 numbers

min (x, y) :- X < y , display (x)
min (x, y) :- X >= y , display (y).

?- min (3, 2).
2.
true

?- min (2, 3)
2
true;
false.

other sols min (x, y, z) :- X < y, z = X.
min (x, y, z) :- X >= y, z is y

?- min (3, 2, z).
R = 2
true.

?- min (2, 3, R), R > 0
R = 2
true.

other sols

min (x, y, x) :- X < y
min (x, y, y) :- X >= y

* Syntax of (if-else):

A -> B ; C

min(x,y) :- X<y -> display(x); display(y).

Example: Write a prolog predicate named compare two numbers.

compare(X,y) :- X>y -> display('greater');
X<y -> display('less');
display(equal)

Ex: if(x>0)
if(x%2==0)
Printf("even");

Sol: is_even(X) :- X>0 -> (X mod 2) =:= 0 -> display(even)

Ex: abs(X,X) :- X>=0

abs(X,y) :- y is -X

?- abs(-2,X)
Y=2.
true.
?- abs(2,X)
X=2
X=-2

modify:
abs(X,y) :- X<0, y is -X

Ex: Write a prolog program to count from X to 10

count(X) :- X > 10 -> display(stopped) ~~greater~~)

~~count(X) : X =<10 & display(X) y is X+1, disp~~

(X =<10 --> y is X+1, display(y).
, count(y).

other sol:

~~c(11).~~

c(X) : X >= 0, X =< y, display(X),
y is X+1, c(y).
c(11). % will not print false.

Ex: count(X, y).
first second.

c(X,y) :- X = Y, display(X).
c(X,y) :- X < y, display(X), z is X+1, c(z, y).

or: c(y,y) :- display(y).

Print/1 and Write/1 predicate
displays text only not expression

Write_In() new line
tab(X) puts X spaces.

read(X) reads data from user, store in X.

Example:

```
Start :-  write('Hello What's Your Name');
          read(X),
          write('Welcome'), tab(2),
          write(X).

?- start.
```

Example: Write a prolog program to read an integer value and print all the numbers from 1 to X, your program should start with query 'go'.

```
go:- write('Enter the number'),
     read(X),
     count(1, X).

count(Y,Y):- write(Y).
count(Y,X):- Y<X, write(Y), Z is Y+1, count(Z,X).
count(Y,Y):- write(Y).

?- go.
```

---

* Assert predicate

assert(X) ; assertz(X) adds to the end

assert(X) adds to the begin

```
?- assert(p(a)).
true
?- p(a)
true
?- assert(p
```

D.B :

p(a)

?- assert(p(b))

error

can't add predicate if it was static

p(a) is static, the user can't modify what the programmer does unless it was dynamic

to solve this :-

D.B :

:- dynamic p/1, r/1.

p(a).

f(a).

r(a)

?- assert(p(g)).

true

?- assert(f(g)).

error.

Retract:

retract(X): removes fact X

retractall(X): removes all facts clauses of y

a(1)
a(2)
a(3)

```
?- retract(a(3))
   true
?- a(X).
   X=2;
   X=3
```

```
?- retract(a(X)).
   X=1;
   X=2;
   X=3.
```

a(1).
a(2).
a(3).

(underscore)

```
?- retract(a(_)).
   true;
   true;
   true.
```

---

a(1). a(2). a(3). a(1,1). a(2,1).

```
?- retractall(a(_)).
   true.
?- retractall(a(_,_))
   true.
```

Example:

```
?- assert(p(a)), asserta(p(b)), retract(p(X)),
   write(X), fail.
b a
false.
```

---

* Cuts:

prevents backtracking:

ex: abs(X, X) :- X >= 0, !.
abs(X, Y) :- Y is -X

```
?- abs(2, X).
X = 2.
```

ex: min(X, Y, X) :- X < Y, !.
min(X, Y, Y) :-

```
?- min(2, 3, R).
R = 2.
?- min(3, 2, R).
R = 2.
```

Ex: c(0) :- write(10), !.
c(X) :- X >= 0, X < 10, write(X), Y is X+1, c(Y).

Ex: a(X) :- b(X). **
a(X) :- c(X).
b(X) :- g(X), h(X). *
b(X) :- X=24, h(X).
g(16).
g(7)
h(X).
c(2).

?- a(X).
~~X=16;~~
~~X=24;~~
~~X=2.~~

X = 16;
X = 7;
X = 24;
X = 2.

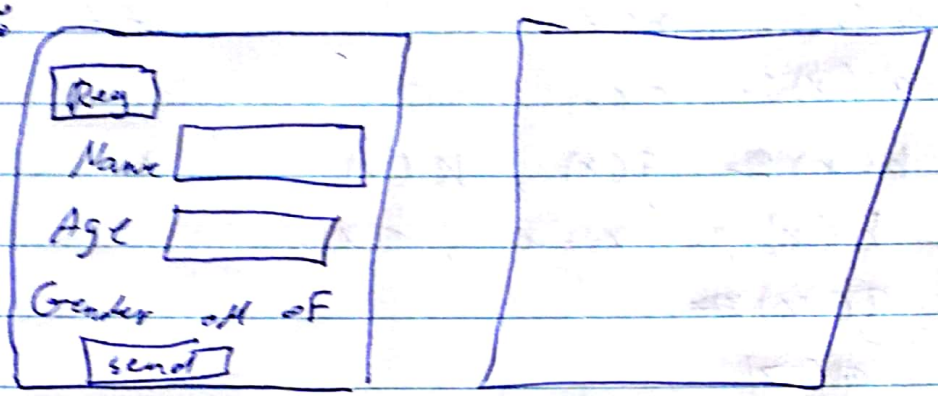
* modify b(X) :- g(X), !, h(X).
?- a(X).
X = 16;
~~X=24;~~
~~X=2.~~

First ex: a(X) :- b(X), !.
X = 16.

*

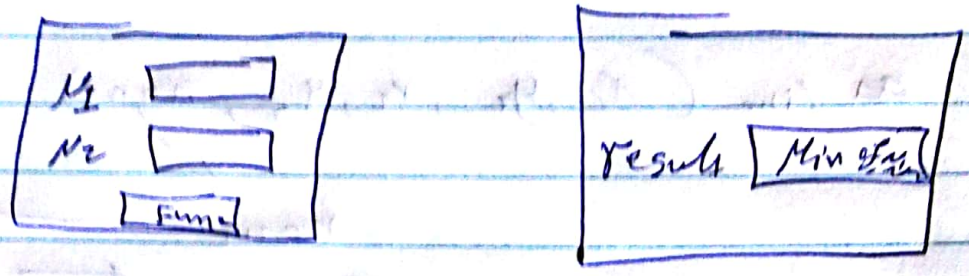
* XPCE toolkit
used for graphical user interface
in Prolog

Ex:



go :- new(D, dialog('Form1')),

Example :




```
show :- new(D, dialog('Form1')),
        send(D, append(new(N1, text_item
                ('N1:')))),
        send(N1, type, int),
        send(D, append, new(N2, text_item('N2:'))),
        send(D, append, button('Find', message(
                @prolog, p, N1?selection,
                N2?selection))),
        send(D, open).

p(X, Y) :- new(D, dialog('Form2')),
        send(D, append, new(T, text_item
                ('Result :'))),
        min(X, Y, R),
        send(T, value, R),
        send(D, open.
min(X, Y, X) :- X < Y !
min(X, Y, Y).
```

Line objects:

Line (x_1, y_1, x_2, y_2, none)

none —

First ←

Second →

both ↔