Chapter-1 Solving Problems by Searching

Steps of problem Solving

1. Goal formulation: Helps us organize the behaviour by limiting the objectives that the agent is trying to achieve.
2. Problem formulation: Given the goal, what actions & states we need to consider.

Search: Sequence of actions that reaches the goal. Search algo takes in a problem as an input returns sol^n ⇒ sequence of actions.

Problem
1. initial state
2. actions: description that are available to the agent
3. a further description of exactly what each action takes (transition model)
4. state space: set of all states reachable from the initial states.
   - this forms a graph with states as nodes & actions as edges.
5. goal test: determines whether a given state is a goal state.
6. path cost: numeric cost of taking each path.

Note: $c(s, a, s')$ = cost of going from s to s' by taking action a = Step cost

Algorithm for tree search:

This doesn't need an EXPLORED ⇩ set. Because here we shall never visit the same node twice.

Tree Search (problem) returns a sol^n or a failure
Initialize the frontier using the initial state of the problem
loop do — frontier ⇐ queue, pq or any DS
- frontier empty or not ⇐ if frontier = empty ⇒ return failure.
- Choose a leaf node & remove it from the frontier ⇒ BFS style.
- goal test ⇐ If node = = goal ⇒ then return the corresponding sol^n.
- expand the node adding the resulting nodes to the frontier to expand.
  - only if it is not already explored.

Note:
In a tree search algorithm it is important to avoid exploring redundant paths. To do this, we augment the tree search algorithm with a data structure called Explored Set (also known as Closed List) which remembers every expanded node.

b → the max^m no. of branches of any node, d → shallowest goal node depth
m → max^m length of any path.

Uninformed Search Algorithms:

↳ No def^n (additional) beyond the problem def^n

① Breadth first search:

Uninformed search algo in which the shallowest unexpanded node is chosen to be expanded

Completeness: [the Algo always finds a sol^n]

optimal when all step costs are equal.

If the shallowest goal node is at some depth d, BFS will eventually find it generating shallower nodes (provided branching factor b is finite). As soon as a goal node is generated we know that all shallower nodes must have been generated already & failed the goal test. [nodes above the given node in the tree]

Complexity Analysis:

BFS is optimal if the path cost is a non-decreasing function of the depth of the node.

If b is the branching factor & we have a uniform tree & let the goal be at depth d then T.C $O(b^{d+1})$ (Summation) $\Rightarrow T = 1 + b + b^2 + b^3 \dots b^d = O(b^{d+1})$ — if you are writing sum in any algo

$S = \underbrace{O(b^{d-1})}_{\text{closed set}} + \underbrace{O(b^d)}_{\text{frontier set}} = O(b^d)$

d → depth of the goal node.
b → branching factor.

Pseudocode:

function BFS (problem) returns a sol^n or a failure (also for UCS)
node ← a node with STATE = problem.~~state~~.initial-state.
INITIAL STATE, PATH-COST = 0
if problem.GOALTEST (node.state) ⇒ return SOLUTION(node)] We don't do this here for UCS
frontier ← a FIFO queue with node as the only element ↳ priority queue for UCS.
explored ← an empty set.
loop do:
    if empty? (frontier) return FAILURE
    node ← POP (frontier) * ↳ here we test for the goal test in UCS.
    ADD node.STATE to explored
    for each action in problem.ACTIONS (node.state) do
        child ← CHILD-NODE (problem, node, action)

if child.STATE is NOT in explored or frontier then:

if GOAL-TEST(child.STATE) ⇒ return SOLUTION(child)

don't check here in UCS.

frontier ← INSERT(child, frontier) in its frontier

↳ if some nodes has higher path cost than child replace it. this step is

↳ relaxation cond"

② UNIFORM COST SEARCH:

This expands the node n with the lowest path cost g(n).

Frontier here is a priority queue ordered by the path cost.

Two differences from BFS:

① Goal Test is applied to a node when it is selected for expansion. rather than when it is first generated.

② A Test is added in case a better path is found to a node currently on the frontier.

Optimality:

In general UCS is optimal. When we observe that whenever UCS selects a node n for exp" the optimal path to that node has been found. Then because the step costs are non-negative paths never get shorter as further nodes are added.

let C^* = cost of optimal sol"

Every action costs atleast ϵ

Completeness is guaranteed provided the cost of every step exceeds the prev. by some +ve cst Δ

$$T = O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$$

When all step costs are same, UCS is similar to BFS but BFS stops as soon as a goal node is generated but UCS examines all nodes at goal's depth to see if one has a lower cost; UCS strictly does more work by expanding nodes at depth d.

③ DEPTH FIRST SEARCH

```
dfs(node):
    visited[node] ← TRUE
    for nbhr : g[node]
        if ! visited[nbhr]
            dfs(nbhr)
```

→ Algorithm for DFS

DFS always expands the deepest node in the current frontier of the search tree.
Now in DFS instead of a queue, we use a LIFO

* Graph search version of DFS is complete while the tree-search version is NOT complete. However, both versions are NOT optimal because they fail if the state space is infinite.

⇒ If the sol" will be the right sub-tree wrt the root, then the entire left-subtree will be explored.

No. of nodes generated = $O(b^m)$ — m: max. depth of any node ($m \geq d$) can be ∞ if state-space can be infinite if the state space is infinite.

T: $O(b^m)$.

S: $O(bm)$

↳ But if we use backtracking search, only one successor is generated at a time rather than all the successors; each partially expanded node remembers which successor to go next. Hence, $S = O(m)$

(4) DEPTH LIMITED SEARCH :

Here we basically apply DFS but with pre-determined depth limit l. This means, the nodes at depth l are treated as if they have no successors. This solves the "infinite" problem.

It is incomplete cuz what if $l < d$ (d is the depth of goal node). Also it is non-optimal if we choose ~~bad~~ $l > d$.

T: $O(b^l)$

S: $O(bl)$

PSEUDO CODE :

```
function DLS (limit, problem)  returns a sol"/ failure or cutoff.
    return recursive-DLS( Node( problem.INITIAL STATE), problem, limit)

function recursive-DLS (node, problem, limit)
    if problem.GOAL-TEST( node.state) then return SOLUTION (node)
    else if limit = 0 then return cutoff.
```

else cutoff_occured ← false
for each action in problem.ACTIONS(node.STATE) do
child ← CHILD-NODE(problem, node, action)
result ← RECURSIVE-DLS(child, problem, limit-1)
if result == cutoff
then cut-off occured ← True.
else if result ≠ failure then
return result ←
if cut_off occured?
return cutoff.
return failure.

⑤ ITERATIVE DEEPENING: Combination of DFS, that finds the best depth limit
M = O(bd) d = depth of goal node.
T = O(b^d) : bd + (d-1)b^2 + ... (1)b^d = O(b^d)
Compared to BFS, it's not very costly.

ALGORITHM:
function IDDFS(problem) returns a sol^n or failure
for depth = 0 to ∞ do:
result ← DLS(problem, depth)
if result ≠ cutoff then
return result.

In general, iterative deepening is the preferred search method when the ~~that~~ search spa is large & the depth of the sol^n is unknown.

⑥ BIDIRECTIONAL SEARCH: Optimal & Complete if both BFS.
Idea behind bidirectional search is to run two simultaneous searches — one forward from the initial state & other backward from the goal — hoping that the two searches meet in the middle
T = O(b^{d/2}) ~ can be optimised by doing using iterative deepening
S = O(b^{d/2}) ~ major weakness

INFORMED SEARCH STRATEGIES:

The general approach we consider here is called Best First Search. It's an instance of the general Tree Search & Graph Search Algo in which a node is selected for expansion based on an evaluation function, $f(n)$.

① Greedy Best First Search:

This algo expands a node that is closest to the goal, on grounds that it'll lead to the solution quickly.

It's incomplete even in finite state space.

$S = O(b^m)$ } But with a good heuristic the complexity can be reduced signifi
$T = O(b^m)$ }

② A* Search Algorithm:

It is a popular form of Best-first search

here evaluation function is $f(n) = g(n) + h(n)$

estimated cost of the cheapest sol^n through n

$g(n)$ = Cummulative path cost from state to node n.

$h(n)$ = Estimated cost of the cheapest path from n to the goal.

A* is both complete & optimal

Identical to UCS except A* uses $h+g$ instead of g.

Conditions for optimality:

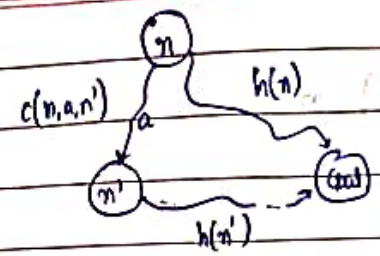
① $h(n)$ should be an admissible heuristic. An admissible heuristic is the one which never overestimates the cost to reach the goal.

⇒ $f(n)$ never overestimates the true cost of a sol^n along the current path through n.

Stricter check

② $h(n)$ should be consistent. $h(n)$ is said to be consistent if for every node n and every successor n' of node n generated via action a, the estimated cost of reaching the goal from n is no longer than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

$$h(n) \leq c(n, a, n') + h(n')$$



Triangle Ineq :

$$h(n) \leq h(n') + c(n, a, n')$$

Every Consistent heuristic is admissible.

* Tree Search version of A^* is optimal if $h(n) \Rightarrow$ admissible, graph search version is optimal if $h(n) \Rightarrow$ consistent

$\Downarrow$

Proof: C-1: if $h(n)$ is consistent then values of $f(n)$ along a path is non-decreasing.

Proof: Triangular ineq :

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

C-2 Whenever A^* selects a node for exp^n, the optimal path to that node has been found.

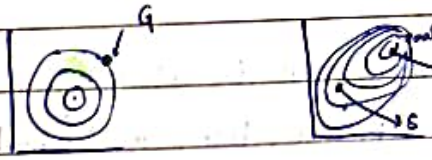
As f is non-decreasing, n' would have a lower f-cost than n & would have been selected first.

C-1 & C-2 prove the optimality

If C^* is the cost of optimal sol^n path

$\Rightarrow$ A^* expands all nodes with $f(n) \leq C^*$

UCS →



$T: O(b^{\Delta})$ $\quad \Delta = h^* - h$, $\epsilon = \frac{h^* - h}{h^*}$

If step cost was const

$T: O(b^{\epsilon d})$ $\quad d \rightarrow$ sol^n depth.

Algorithm for RBFS of A^*

Again

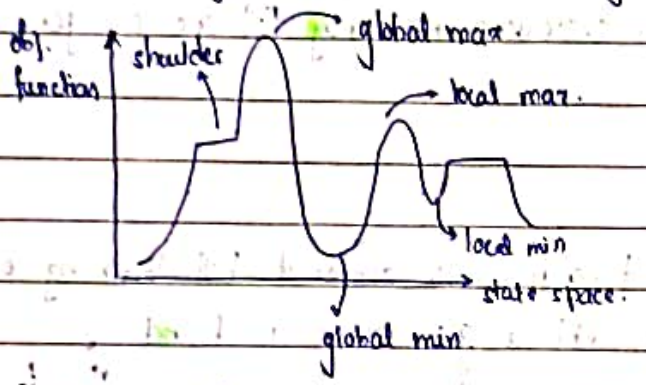
```
function RBFS (problem):
    return helper (problem, Node (initial-state), ∞)

function helper (problem, node, f-limit):
    if problem.GoalTest (state)
        return Solution (node)
    successors ← []

    fo each action in Actions (node.state) do:
        add Child-Node (problem, node, action) into successors.

    if successors == ϕ
        return failure
    fo each s in successors:
        s.f ← max (node.f, s.g + s.h)

    loop do:
        best ← lowest f-value node in successors.
        if best.f > f_limit
            return failure
        alt ← second-lowest f-value node in successors
        result, best.f ← helper RBFS (problem, best, min (f_limit, alt))
        if result ≠ failure
            return result.
    return failure.
```

Chapter - 2: Local Search.

Local Search Algorithms are used when we doesn't need to worry about paths at all. Generally these algos operate using a single node & we move only to neighbours of that node.

Advantages:

(i) They use very little memory (usually constant)
(ii) They often find reasonable sol's in large or infinite state spaces.
(iii) local Search Algos are best for finding the best state acc. to some objective function.



① HILL CLIMBING: Greedy local search.

Pseudocode:

function HILL-CLIMBING (problem):

    current ← MAKE-NODE (problem. INITIAL-STATE)

    loop do:

        neighbour ← highest valued successor of current

        if neighbour. value ≤ current. value

            return current. STATE

        current ← neighbour.

It's a loop running in the dir. of uphill, when it reaches a peak, it stops!!
It may get stuck cuz of:

① Local maxima: A peak < global maxima.
② Ridges: Ridges result in a sequence of local maxima that is very difficult to greedy algos to navigate.
③ Plateaus: It's a flat area. hill climbing search may get all stuck there.

It gets stuck 86% of the times.

⇒ When it gets stuck in a shoulder & if we always allow sideways moves, then when there are no uphill moves, it'll get stuck in an infinite loop. One sol" to this is to put up an upper limit to the no of consecutive side-ways moves. This raises the success % to 94%

Variants:

① Stochastic hill-climbing: chooses a random uphill moves from several possible moves, the probability can vary with steepness. But it converges slowly compared to steepest ascent

② first-choice hill-climbing: Generates successors randomly till one is generated that is better than the current state.

These are all in-complete.

③ Random-restart hill climbing: It conducts a series of hill-climbing searches from randomly generated initial states, until a goal state is found.

If p is prob of success, no. of restarts = 1/p

This is trivially complete since at some pt the goal state will be (initial state)

If there are a few local minima & plateaux then it can find a sol" quickly.

④ Simmulated Annealing:

The innermost loop of simmulated annealing is similar to hill climbing. Instead of picking the best move here, instead here we pick a random move. If the move is good (improves the situation), it is accepted. Otherwise, the algo accepts the move with a p

This probability ↓ exponentially with the "badness" of the move. This prob also as T (↓). So a bad move is allowed initially but might not be allowed Temperature T (↓).

PSEUDOCODE: a mapping from time to temperature

function Simmulated-Annealing (problem, schedule):

current ← Make-Node (problem.initial-state)

for t ← 1 to ∞ do:

remember this !!

T ← schedule(t)

if T == 0

return current

next ← randomly selected successor of current. ← first choice hill climbing

ΔE ← next.value − current.value

if ΔE > 0

current ← next.

else

current ← next only with probability $e^{\Delta E/T}$.

③ Local k-beams search :

Local k-beams search keeps track of k states (parallel on k threads). It begins with k-randomly generated states, if any one of them is a goal state, then algo halts. Otherwise it selects the k best successors from the complete list & repeats.

k random in case of stochastic k-beams algo.

④ Genetic Algorithms: (Variant of Stochastic beam search - uses two parent states instead of one)

Pseudocode:

measures the fitness of individual.

function Genetic-Algo (population, fitness)

repeat

new_population ← empty_set

for i = 1 to SIZE(population) do:

x ← random-selection (pop^n, fitness)

y ← random-selection (pop^n, fitness)

child ← Cross-over (x, y)

if (small_random_probability)

child ← Mutate(child)

add child to new_population

until some individual is fit enough a sufficient time has gone.

return the best found individual acc. to fitness score.

c ← rand. int from 1 to n

n ← length(x)

return

Append(Substr(x, 1, c), Substr(y, c+1, n))

Chapter-3: Adversarial Games

When the goals of agents are in conflicts wrt each other, then such problems are called Adversarial Search problems (or Games)

Zero Sum Games: Utility values at the end are equal & opposite.
(eg: In chess, if one wins (+1) then the other loses (-1))

Pruning: Ignoring the portions of the search tree that make no difference to the final choice, and heuristic evaluation functions help us approximate the true utility.

Parts / Structure of a Game:

1) S_0 : Initial State.
2) Player(s) : Defines which player has the move in a state.
3) Action(s) : The transition model which defines the result of a move.
4) Terminal-test(s): Test which indicates if the game is over (returns true if so).
5) Utility(s, p): It determines a numeric value of the game for a player p, ending at a terminal state s.

Game Tree: A tree with nodes ⇒ game states & edges ⇒ moves.
In a game tree, optimal soln would be a sequence of actions leading to a goal state.

Minimax Algorithm:
Minimax value of each node is the utility (for MAX) of being in the current-state, assuming that both players play optimally till the end of the game.

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s) & \text{if Terminal-Test}(s) \text{ is True.} \\ \max_{a \in \text{Actions}} \text{Minimax}(\text{Result}(s,a)) & \text{if } P = \text{MAX} \\ \min_{a \in \text{Actions}} \text{Minimax}(\text{Result}(s,a)) & \text{if } P = \text{MIN.} \end{cases}$$

Algorithm: (Performs Depth-first exploration of the tree)
T: $O(b^m)$ m: max depth, b: no of legal moves at a point
S: $O(bm)$ If all actions are generated once, $O(m)$ for an alg that generates one action at a time.

function minimax(state) returns an action:
return argmax $_{a \in Actions}$ Min-value(state Result(state, a)).

function max-value(state) returns a utility value:
if terminal-test(state)
return Utility(state)
$v \leftarrow -\infty$
for each a in Actions(state) do:
$v \leftarrow$ Max(v, min-value(Result(state, a)))
return v.

function min-value(state) returns a utility value:
if terminal-test(state)
return Utility(state)
$v \leftarrow \infty$
for each a in Actions(state)
$v \leftarrow$ min(v, Max-value(Result(state, a)))
return v.

If the game is NOT zero then some kind of alliances might also be formed.

Alpha-Beta Pruning:
General approach: Consider a node n, such that player has a choice of moving to that node. But if the player has a better choice m either at the parent of the node or at any choice further up, then n will never be reached in actual play.

test:
α: the value (highest) we have found so far at any choice point along the path for MAX
β: the best value (lowest) we have found so far at any choice pt. along the path for MIN.

The success of this pruning depends on the order in which the traversal is done.
At random, nodes traversed = $O(b^{3m/4})$

classmate
Date
Page

function α-β-search (state) returns an action

$v \leftarrow$ Max-Value (state, $-\infty$, $+\infty$)

return the action in Actions (state) with value v

function Max-Value (state, α, β) returns a utility value

if Terminal-Test (state)

return Utility (state)

$v \leftarrow -\infty$

for each a in Actions (state)

$v \leftarrow$ Max (v, Min-Value ((Result(s,a), α, β))

if $v \geq \beta$

return v // Prune if.

$\alpha \leftarrow$ Max (α, v)

return v.

function Min-Value (state, α, β) returns a utility value.

if Terminal-Test (state)

return Utility (state)

$v \leftarrow +\infty$

for each in Actions (state)

$v \leftarrow$ Min (v, Max-value (Result(s,a), α, β)).

if $\alpha \geq v$

return v.

$\beta \leftarrow$ Min (β, v)

return v.

Norvig Neural Network Notes

Univariate linear Regression:

Here $y = w_1 x + w_0$ let w is a vector $[w_0, w_1]$

We need to find weights w_0 and w_1 that minimize the empirical loss.

$$\text{loss} = \sum_{j=1}^{N} L_2(y_j, h_w(x_j)) = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

This sum is minimized over the partial of & w_0:

We now have

$$w_1 = \frac{S_{xy}}{S_{xx}} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - x)^2}$$

$$w_0 = \bar{y} - w_1 \bar{x}$$

where $\bar{x}$ and $\bar{y}$ are means of all x & y values.

If graph the loss function, then we can observe that the loss function is convex. (True for every SLR which uses L_2 norm)

So alternatively to obtain the local minima of the loss function we can use gradient descent which is stated as follows:

$w \longleftarrow$ any point in the parameter space

loop until convergence do:

for each w_i in $\vec{w}$ do:

$$w_i \longleftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{loss}(\vec{w})$$

Here, the update eq'n will be:

$w_0 \longleftarrow w_0 + \alpha (y - h_w(x))$ ← predicted value

$w_1 \longleftarrow w_1 + \alpha (y - h_w(x)) \times x$

↳ $\frac{1}{m} \sum$ for m training examples.

Learning Rate: can be fixed, const or can decay over time as the learning happens

We can use batch gradient descent (Convergence is slow but guaranteed for some α) or we can use stochastic gradient descent techniques (convergence is fast but not guaranteed) so simulated annealing also helps

Multivariate linear Regression:

$h_w(x_j) = w_0 + w_1 x_1 + w_2 x_2 \dots w_n x_n$ ← dummy var.

$= w_0 x_j$

$= w^T x_j$

$= \sum w_i x_{j,i}$

Same as before we apply squared error loss function to obtain $\vec{w}$.

Analytic sol^n:

$$w^* = (X^T X)^{-1} X^T y$$

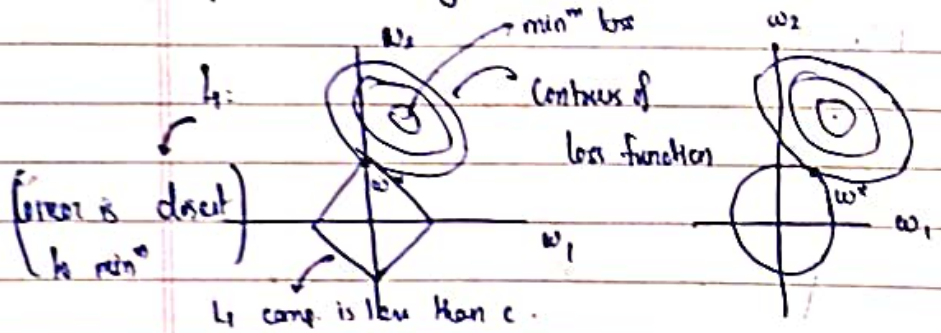
It is common to use regularization on MLR to prevent overfitting.

$$\text{Cost}(h) = \text{Emp loss}(h) + \lambda \text{ Complexity}(h)$$

$$\text{Complexity} = L_q(w) = \sum_i |w_i|^q$$

$q=1$: L_1 regularization. ⇒ it tends to produce a sparse model ↓ sets many wts to 0

$q=2$: L_2 regularization:



for L_1 : min. available loss (concentric contours) occurs along the axis while L_2 loss regularization, the minimal loss can occur anywhere on the circle ⇒ no preference to zero weights.

L_1 : rotationally variant

L_2 : rotationally invariant

Linear Classifiers:

Decision Boundary: line that separates two classes.

$h_w(x)$ = Threshold $(w.x)$ where Threshold$(z) = 1$ if $z \geq 0$ else 0.

Update rule = $w_i \longleftarrow w_i + \alpha\left(y - h_w(x)\right) \times x_i$

This is called Perceptron learning rule.

In general, the perceptron learning rule may not converge to a stable sol^n for a fixed learning rate α, but if α decays as $O(1/t)$ where t is the no. of iterations, the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence.

New Threshold = Logistic $(z) = \frac{1}{1+e^{-z}}$ → $g(z)$.

↓

forms a soft boundary in the input space → 0.5 for middle of boundary, 0 or 1 as it approaches away from the boundary.

Here $\frac{\partial}{\partial w_i} \text{Loss}(w) = \frac{\partial}{\partial a_i} \text{Loss}(w) \times \frac{\partial a_i}{\partial w_i}$

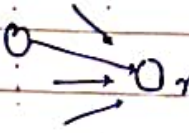
$= g'(w.x) \times (-2(y - h_w(x)))\, x_i$

$= -2\, h_w(x)\,(1 - h_w(x))\,(y - h_w(x))\, x_i$

So

$$\boxed{w_i \longleftarrow w_i + \alpha\,(y - h_w(x)) \times h_w(x)\,(1 - h_w(x))\, x_i}$$

Neural Networks:



$$\boxed{in_j = \sum_{i=0}^{n} w_{i,j}\, a_i}$$

$a_0 = 1$

$$\boxed{a_j = g(in_j)}$$

$g \rightarrow$ Activation function.

A neural network in which all inputs are connected directly to the outputs, is called a single-layer neural network.

Date
Page

more than

Note that: The majority function which outputs a 1 only if half of its inputs can be represented as a perceptron with $w_i = 1$ & $w_0 = -n/2$.

A single perceptron can represent: AND, OR and NOT.

first O/P layer $\Delta_k = Err_k \times g'(in_k)$. $y - h_w$

at k^{th} unit of the O/P layer.

~~$W_{j,k} \leftarrow W_{j,k} + \alpha \times a_j \times \Delta_k$~~ → activation value of before layer. **Ignore this !!**

hidden layer $\Delta_j = g'(in_j) \times \sum_k w_{j,k} \Delta_k$.

update weights

$\Delta w_{i,j} \leftarrow \alpha \times a_i \times \Delta_j$

Refer iPad note for H

Algorithm : (ex, network)
function backProp, returns a neural network.

repeat

for each $w_{i,j}$ in network do :

$w_{i,j} \leftarrow$ a small random no.

for each example (x, y) in examples do :

Initialisation & starting weights

for each node i in the input layer do :

$a_i \leftarrow x_i$

Forward Propagation

for $l = 2$ to L :

for each node j in layer l do :

$in_j \leftarrow \sum_i w_{i,j} a_i$

$a_j \leftarrow g(in_j)$

Back-propagation

o/p layer

for each node j in the o/p layer do :

$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$

hidden layers

for $l = L-1$ to 1 do :

for each node i in layer l do :

$\Delta[i] \leftarrow g'(in_j) \sum_i w_{i,j} \Delta[j]$

updation of weights.

for each weight $w_{i,j}$ in network do :

$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$

until some stopping criteria is reached.

return network.

Book involving back propagation :

O/P layer: $\frac{\partial Loss_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}}$.

$$= -2(y_k - a_k)\, g'(in_k)\frac{\partial in_k}{\partial w_{j,k}}$$

$$= -2(y_k - a_k)\, g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k} a_j\right)$$

$$= \underbrace{-2(y_k - a_k)\, g'(in_k)}_{\Delta}\, a_j$$

$$= -a_j \Delta_k$$

Hidden Layers (find from the end.)

$$\frac{\partial Loss_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}}$$

$$= -2(y_k - a_k)\, g'(in_k)\frac{\partial in_k}{\partial w_{i,j}}$$

$$= -2\Delta_k \frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}\, a_j\right)$$

$$= -2\Delta_k\, w_{j,k}\frac{\partial a_j}{\partial w_{i,j}}$$

$$= -2\Delta_k\, w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}}$$

$$= -2\Delta_k\, w_{j,k}\, g'(in_j)\frac{\partial in_j}{\partial w_{i,j}}$$

$$= -2\Delta_k\, w_{j,k}\, g'(in_j)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j} a_i\right)$$

$$= \underbrace{-2\Delta_k\, w_{j,k}\, g'(in_j)}_{\Delta_j}\, a_i$$

$$= -a_i \Delta_j$$

Perceptron bound Mistakes.

Hyperplane :

$$H = \{ x \mid w^T x = b \}.$$

Further

$$H = \{ x \mid \theta^T x = 0 \text{ and } x_0 = 1 \}$$

where $\theta = [b, w_1, \ldots w_n]$.

Half Spaces :

$$H^+ = \{ x \mid \theta^T x > 0 \text{ and } x_0 = 1 \}$$

$$H^- = \{ x \mid \theta^T x < 0 \text{ and } x_0 = 1 \}.$$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

(Online) Perceptron Algo.

$\theta \leftarrow 0$

for $i \in \{1, 2, \ldots\}$ do :

$\hat{y} \leftarrow \text{sgn}(\theta^T x^{(i)})$

if $\hat{y} \neq y^{(i)}$ then

$\theta \leftarrow \theta + \alpha(y^{(i)} - \hat{y}) x^{(i)}$

return θ.

In case of batch training one extra outer loop to check for convergence

$x^{(i)} y^{(i)}$ acc. to the slides.

Extensions on a perceptron :

→ Voted Perceptron. (Memory intensive. Keeps around every weight vector seen during training).

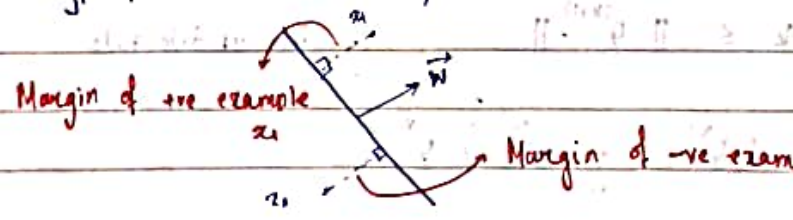
→ Averaged Perceptron. (Same as voted perceptron but can be implemented in an eff. way).

→ Kernel Perceptron (Apply the kernel trick $K(x', x)$ to the perceptron)

→ Structured Perceptron (Basic idea can also be applied when y ranges over an exponentially large set)

Geometric Margin.

The margin of an example x_i wrt a linear separator w is the dist of the plane or hyperplane $w^T x = 0$ from x_i



⇒ Margin γ_w of a set of examples S wrt l.s w is the smallest margin over all points $x \in S$

⇒ The margin γ of a set of examples S is the max γ_w over all l.s.

Set of pts → if there exist a linear boundary ⇒ linearly separable.

Theorem: If data has margin γ & all points inside a ball of radius R, then the perceptron makes $\leq (R/\gamma)^2$ mistakes.

Note: We say that the batch perceptron algo has converged if it stops making mistakes on the training data.

Cond's: 1. $\|x^{(i)}\| \leq R$

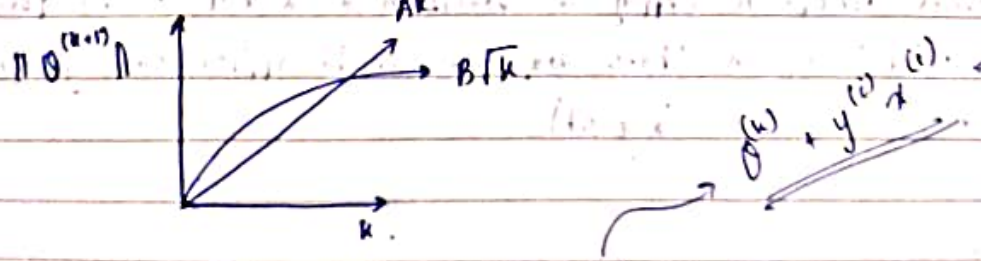
2. $\exists\ \theta^*$ s/t $\|\theta^*\| = 1$ and $y^{(i)}(\theta^* \cdot x^{(i)}) \geq \gamma\ \forall i$ ← Margin.

Then no. of mistakes made by the perceptron in the dataset $\leq (R/\gamma)^2$. ← k.

Proof:

To show that there exists A & B s/t

$$Ak \leq \|\theta^{(k+1)}\| \leq B\sqrt{k}.$$



⇒ Use the prev algo keep a counter: $\theta^* \theta^{(k+1)}$

To prove: $Ak \leq \|\theta^{(k+1)}\|$ ← for update rule.

$$\theta^{(k+1)} \cdot \theta^* = (\theta^{(k)} + y^{(i)} x^{(i)})\, \theta^*$$

classmate
Date
Page

separate.

$$= \theta^{(k)} \cdot \theta^* + y^{(i)} \cdot (\theta^* \cdot x^{(i)})$$

$$\geq \theta^{(k)} \cdot \theta^* + \gamma$$

$y^{(i)} (\theta^* (x)^{(i)}) \geq \gamma$. by def^n.

$\Rightarrow \theta^{(k+1)} \cdot \theta^* \geq k\gamma$ by induction on k ($\because \theta^{(1)} = 0$)

$\Rightarrow \| \theta^{k+1} \| \geq k\gamma$. (as $\|w\| \times \|u\| \geq w^T u$. and $\|\theta^*\| = 1$)

Cauchy Schwartz.

& more : $\| \theta^{(k+1)} \| \leq R\sqrt{k}$

$$\| \theta^{(k+1)} \|^2 = \| \theta^{(k)} + y^{(i)} x^{(i)} \|^2$$

$$= \| \theta^{(k)} \|^2 + (y^{(i)})^2 \| x^{(i)} \|^2 + 2y^{(i)} (\theta^{(k)} \cdot x^{(i)})$$

$$\leq \| \theta^{(k)} \|^2 + (y^{(i)})^2 \| x^{(i)} \|^2$$

as this is ≤ 0.

$$= \| \theta^{(k)} \|^2 + R^2. \quad (\because (y^{(i)})^2 \| x^{(i)} \|^2 = \| x^{(i)} \|^2 = R^2 \text{ as } (y^{(i)})^2 = 1)$$

$$\Rightarrow \| \theta^{(k+1)} \|^2 \leq kR^2. \quad (\text{by induction also as } (\theta^{(1)})^2 = 0)$$

$\Rightarrow \| \theta^{(k+1)} \| \leq \sqrt{k} R$

So, we have:

$$\sqrt{k}\gamma \leq \sqrt{k} R$$

$$\Rightarrow \boxed{k \leq (R/\gamma)^2}.$$

Theorem:

let $d_i = \max \{ 0, \gamma - y_i (u \cdot x_i) \}$

Define $D = \sqrt{\sum_{i=1}^{m} d_i^2}$

Then $\boxed{k \leq \left(\frac{R+D}{\gamma} \right)^2}$.

Data Clustering

Partitional Clustering Algorithms:

K-Means Clustering: One of the most widely used partitional clustering algorithms. It is a greedy algorithm which is guaranteed to converge to a local minimum but the minimization of its score function is NP-Hard.

Algorithm:

Select K points as initial clusters

repeat

Form K clusters by assigning each point to its closest centroid.

Recompute the centroid of each cluster

until convergence criterion is met

Proximity measures ⟹

L_1 Norm, L_2 Norm, Cosine Similarity

Given $D = \{x_1, x_2, \ldots x_N\}$

Aim is to return $C = \{C_1, C_2, \ldots C_k\}$

s.t.

$$\text{Sum of sq. errors } (C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - C_k\|^2$$ ← L_2 Norm

is minimum.

Assigning.

$$C_k = \frac{\sum_{x_i \in C_k} x_i}{|C_k|}$$ → Updating

C_k is the mean of the k^{th} cluster. $|C_k|$ = no. of points in C_k.

$$SSE(C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} (C_k - x_i)^2$$

$$\frac{\partial}{\partial C_j} SSE = \sum_{x_i \in C_j} 2(C_j - x_i) = 0$$

$$\Rightarrow 2|C_j| C_j - 2\sum x_i = 0$$

$$\Rightarrow C_j = \frac{\sum_{x_i \in C_j} x_i}{|C_j|}$$

Sum of all points / No. of points in the cluster.

Here with each iteration, SSE monotonically decreases. Hence it converges to a local min^m.

Natural Algorithms.

1. Particle Swarm Optimization Algorithm.

PSO Search Strategy: In each iteration we need to update the personal best sol^n, team best sol^n & the current direction of movement.

Analogy: (n-dim state space)

Team Member A: $\vec{X}_1^d = [x_1^d, y_1^d]$

Team Member B: $\vec{X}_2^d = [x_2^d, y_2^d]$

Team Member C: $\vec{X}_3^d = [x_3^d, y_3^d]$

$\Big\}$ $\vec{X}_i^d = [x_i^d, y_i^d, z_i^d, \dots]$ in general for i^{th} team member

$$\vec{V}_i^{d+1} = 2r_1 \vec{V}_i^d + 2r_2(\vec{P}_i^d - \vec{X}_i^d) + 2r_3(\vec{G}^d - \vec{X}_i^d)$$

- $\vec{V}_i^{d+1}$: Next velocity (for tomorrow)
- $2r_1 \vec{V}_i^d$: (Current velocity)
- $2r_2(\vec{P}_i^d - \vec{X}_i^d)$: Personal best position. Distance to the personal best
- $2r_3(\vec{G}^d - \vec{X}_i^d)$: Global best sol^n. Distance to the global best

r_1, r_2 and r_3: Random no.s to increase or decrease the impact.

Also we have,

$$\vec{X}_i^{d+1} = \vec{X}_i^d + \vec{V}_i^{d+1}$$

- $\vec{X}_i^{d+1}$: Next position on next day
- $\vec{X}_i^d$: Position of day d
- $\vec{V}_i^{d+1}$: Velocity in day d+1.

Mathematically we have:

$$\vec{X}_i^{t+1} = \vec{X}_i^t + \vec{V}_i^{t+1} \quad \text{(Update position)}$$

$$\vec{V}_i^{t+1} = w\vec{V}_i^t + c_1 r_1(\vec{P}_i^t - \vec{X}_i^t) + c_2 r_2(\vec{G}_i^t - \vec{X}_i^t)$$

- $w\vec{V}_i^t$: Inertia
- $c_1 r_1(\vec{P}_i^t - \vec{X}_i^t)$: Cognitive, Individual Component
- $c_2 r_2(\vec{G}_i^t - \vec{X}_i^t)$: Social.

$0.9 \rightarrow 0.1$ sometimes (tuning).

Pseudocode:

Initialize the controlling params (N, c1, c2, wMin, wMax, vMax, MaxIter).
Initialize the population of N particles
do:
  for each particle
    Calculate the objective of the particle
    Update the personal best if required.
    Update the global best if required.
  end for
  Update the inertia weight (w)
  for each particle:

$$\vec{V}_i^{t+1} = w\vec{V}_i^t + C_1 r_1 (\vec{P}_i^t - \vec{X}_i^t) + C_2 r_2 (\vec{G}_i^t - \vec{X}_i^t)$$

$$\vec{X}_i^{t+1} = \vec{V}_i^{t+1} + \vec{X}_i^t$$

  end for.
while the end condition is NOT satisfied.
Return the Global best as the best estimation of the global optimum.

Balancing c_1, c_2 we can change the levels of exploration & exploitation.

Ant Colony Optimisation Algorithm:
Intuition: We travel along all paths. if an ant finds a shorter path then it releases more pheromones along the way causing other ants to follow that path.

$$P_{ij} = \frac{T_{ij}^{\alpha}\, \eta_{ij}^{\beta}}{\sum_{\text{allowed}} T_{im}^{\alpha}\, \eta_{im}^{\beta}}$$

P_{ij}: Prob. of moving from i^{th} city to j^{th} city.
Desire of moving from i^{th} city to j^{th} city
T_{ij}: Amount of Pheromone b/w i^{th} & j^{th} city
η_{ij}: Proximity b/w i^{th} & j^{th} city.

Next we update the pheromones function:

$$\Delta T_{i,j,k}(t) = \begin{cases} \frac{Q}{L_k(t)} & \text{if } (i,j) \in T_k(t) \\ 0 & \text{if } (i,j) \notin T_k(t) \end{cases}$$

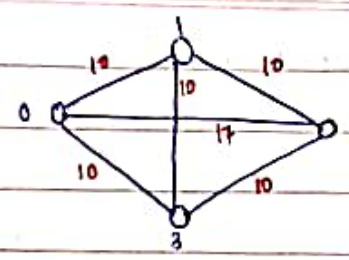
$$T_{ij}(t+1) = (1-\rho) \cdot T_{ij}(t) + \sum_{k=1}^{m} \Delta T_{ij,k}(t)$$

ρ: evaporation rate

classmate
Date
Page

amount of pheromone left → 0.64

Ex:



	Route	Length	Pheromone.
1st ant :	0-1-3-2-0	47	4/47 = 0.085 *
2nd ant :	1-3-0-2-1	47	4/47 = 0.085
3rd ant :	2-3-0-1-2	40	4/40. 0.1 .

Initially per path we had 0.32 pheromones.

ex. 0.64 gm.

Pseudocode:

Initialize pheromone function.

Repeat till convergence :

for all ants i : construction sol (i).;

for all ants i : global pheromone update (i);

for all ant edges : evaporate pheromone

$$T_{i-j} := (1-\rho)\, T_{i-j}$$

Construct Solution (i)

Initialize ant

While not yet a solution :

Expand the sol" by one edge probabilistically acc. to the pheromone.

$$\frac{T_{e,u}}{\sum_{e} T_{e,v}}$$

Global Pheromone update (i) :

for all edges in the solution :

Increase the pheromones acc. to their quality :

$$\Delta T_{j-j} := \frac{Q}{\text{length of the path stored.}}$$

Artificial Bee Colony Algorithm :

The movement of the bees is recorded in three phases : Employed, Onlooker & Scout Phase

① Employed : Here we generate a new sol", calculate a new fitness value & compare it with old sol"

② Onlooker : Calculate probabilities
Produce a new sol" depending on the prob.
Calculate a new fitness and apply greedy selection.

③ Scout Phase: find the abandoned sol" and generate a new sol" randomly to replace them.

Pseudo code: swarm_size, num_iters, dim, lim, employed, onlooker, food source = N/2

for each food source: value of function

Select the update variable and partner: any other variable.

Update the variable: $X_{new} = X + \phi(X - X_p)$, $\phi \in [-1, 1]$ ↳ random no.

Apply Greedy Selection.

if sol" could not improve ⇒ improve the trial = counter.

Calculate Probabilities → $\frac{fit_i}{\sum fit_i}$ (Waggle dance)

Select a random no. (r)

If r < prob: ; new sol" creation

update the food sources based on the random variables & probs.

If trial_counter > limit

generate a new sol" randomly

iter ← iter + 1

Same thing in a refined way:

Explored
1) Initialize the population of candidate sol's. Each position represents a possible sol" to an obj. function. These positions are equivalent to food source located by scout bees.
2) Accessing the quality of food using the following formula: $fit_i = \begin{cases} \frac{1}{1+f_i} & \text{if } f_i \geq 0 \\ 1 + abs(f_i) & \text{if } f_i < \end{cases}$
3) Update the positions of bees: $v_{ij} = x_{ij} - \phi_{ij}[x_{ij} - x_{kj}]$
4) Based on the update values, we replace the old pos if the new pos is better.

5) $P_i = \frac{fit_i}{\sum_{n=1}^{SN} fit_n}$, the food source with high P_i is selected.

↳ Something similar to mutation & does (limit thing).

Logic

Propositional Sentence : Statements that take on True or False values

Not : $\neg$ (Negation)

And : $\wedge$ (Conjunction)

or : $\vee$ (Disjunction)

(Inclusive or (both can happen)

$\rightarrow$: if ... then (implication)

$\leftrightarrow$: if and only if (Equivalence)

Note that :

B if A $\equiv$ if A then B $\equiv$ A only if B

$(\neg A) \vee B$: Not A unless B

So, $A \rightarrow B$ is same as $(\neg A) \vee B$

p	q	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	T	F	F	F
F	T	T	F	T	F
F	F	F	F	T	T

Propositional formulas :

They are expressions involving variables, propositional connectives & parenthesis s.t

a) p_i is a propositional formula, for any $i \geq 1$

b) If A is a propositional formula, then so is $\neg A$

c) If A and B are propositional formulas, then so $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ & $(A \leftrightarrow B)$

These are formal rules but we can be easy!!

Note: $p_1 \wedge p_2$ is NOT a propositional formula. (Missing parenthesis)

$(\neg p_3)$ is NOT a propositional formula (Extra parenthesis)

$p \vee 7$ is NOT a propositional formula (Use of nos other than p_i's)

Precedence of Operators: $\neg > \wedge = \vee > \rightarrow = \leftrightarrow$

If parenthesis are omitted against connectives of the same precedence, they are associated from R to L

$\rightarrow$ is NOT associative.

If C is a propositional formula then there is a unique way to express it. This called the unique readability property.

The general framework for defining a function $h(A)$ by recursion is:

(a) For the base case, we define the value of h for a propositional formula p_i.

(b) We use $h(A)$ to define the value of h for $\neg A$.

(c) Let $h(A)$ and $h(B)$ are defined then we use them to define $h(A \circ B)$ where $\circ$ can be $\wedge, \vee, \rightarrow, \leftrightarrow$

The value of h can uniquely defined by (a) – (c).

Def^n: A truth assignment ϕ is a mapping: ← it is a true row of a truth table.

$$\phi : \{p_1, p_2, \ldots\} \rightarrow \{T, F\}$$

Def^n: Suppose ϕ is a truth assignment (Recursion usage)

(a) If A is $\neg B$ then $\phi(A) = \begin{cases} F & \text{if } \phi(B) = T \\ T & \text{otherwise.} \end{cases}$

(b) If A is $(B \vee C)$ then $\phi(A) = \begin{cases} T & \text{if } \phi(B) = T \text{ or } \phi(C) = T \\ F & \text{otherwise.} \end{cases}$

(c) If A is $(B \wedge C)$ then $\phi(A) = \begin{cases} T & \text{if } \phi(B) = T \text{ and } \phi(C) = T \\ F & \text{otherwise.} \end{cases}$

(d) If A is $(B \rightarrow C)$ then $\phi(A) = \begin{cases} T & \text{if } \phi(B) = F, \text{ or } \phi(C) = T \\ F & \text{otherwise.} \end{cases}$

(e) If A is $(B \leftrightarrow C)$ then $\phi(A) = \begin{cases} T & \text{if } \phi(B) = \phi(C) \\ F & \text{otherwise.} \end{cases}$

Def^n: A formula A is a Tautology if $\phi(A) = T$ for all truth assignments ϕ.

ϕ Input, $p_1, p_2 \ldots$ ex.

φ → arrangement (classmate each row)

A → formula.

In this case, A is also tautologically valid.

Def": A formula A is satisfiable if there is some assignment ϕ such that $\phi(A) = T$.

$\models A$ means A is a Tautology.

eg: $p_1 \vee \neg p_1$ and $p_1 \rightarrow (p_2 \rightarrow p_1)$ are Tautologies

$p_1 \wedge \neg p_1$ is unsatisfiable.

Def": A set Γ (→ set of expressions) of propositional formulas is satisfiable if there is a truth assignment ϕ s/t $\phi(A) = T$ for every $A \in \Gamma$. for such a ϕ we can say that ϕ satisfies Γ

for the example:

Def": let A and B be formulas and Γ be a set of formulas, then:

(a) $A \models B$ if every truth assignment that satisfies A (gives atleast one true value.) also satisfies B (does the same for B)

logically implies

(b) $\Gamma \models B$ if every truth assignment that satisfies Γ (satisfies all the $A \in \Gamma$) also satisfies B.

The cond" $\models B$ holds iff B is a tautology

Deeper: $A \models B$ if & only if $A \rightarrow B$ is a tautology

(A tautologically implies B.)

Theorem: Note: If Γ is finite, $\Gamma = \{A_1, A_2, \ldots A_k\}$, then Γ is satisfied by ϕ if and only if ϕ satisfies $A_1 \wedge A_2 \wedge A_3 \ldots \wedge A_k$.

for all φ if $\varphi(B) = T$ for all $B \in \Gamma$, then $\varphi(A) = T \Rightarrow \Gamma \models A$.

In other words. (this means that A should also belong to Γ)

to all φ (rows), if φ satisfies Γ, then φ also satisfies A

$\Rightarrow \Gamma \models A$ (meaning).

Note:

$\models A$ means that Null $\models A$ (Empty Set)

The cond" $\models A$ holds true if and only if A is a tautology.

We write $A \not\models B$ as $A \models B$ is false.

$\psi \Rightarrow$ assignment

Theorem: Suppose Γ and Δ be sets of formulas s/t $\Gamma \subseteq \Delta$ } Proof: follows from the definitions.

(a) If ψ satisfies Δ, then ψ satisfies Γ ⇝ subset property.

(b) If Δ is satisfiable, then Γ is also satisfiable.

(c) If $\Gamma \models A$ then $\Delta \models A$.

↓ subset property.

Theorem:

(a) let A and B be formulas. Then $\{A, \neg A\} \models B$; unsatisfiable formula.

(b) Let B be a formula and Γ be an unsatisfiable set of formulas. Then $\Gamma \models B$.

Proof:

(a) is a special case of (b).

Part (b): There is no truth assignment satisfying Γ hence it trivially holds $\Gamma \models B$

Def^n: We write $A \models\models B$ if $A \models B$ and $B \models A$.

A and B are said to be tautologically equivalent.

$A \models\models B$ holds if and only if $\psi(A) = \psi(B)$ for all truth assignments.

Semantic Deduction Theorem:

Let A and B be formulas and Γ be a set of formulas, Then:

(a) $A \models B$ if & only if $A \rightarrow B$ is a tautology.

(b) $\Gamma, A \models B$ if and only if $\Gamma \models A \rightarrow B$

↳ $A \notin \Gamma$

Proof: (a) is a special case of (b) when Γ = empty set.

Using def^{n}s $\Gamma, A \models B$ means that for any truth assignment ψ, if ψ satisfies $\Gamma \cup \{A\}$ then $\psi(B) = T$

⇒ if ψ satisfies Γ then and $\psi(A) = T, \psi(B) = T$ ⇝ then

⇒ this is equivalent to " if ψ satisfies Γ then $\psi(A \rightarrow B) = T$

So finally we have $\Gamma \models A \rightarrow B$.

Theorem: Supose Γ is a set of formulas s/t $\Gamma = \{A_1, A_2, \ldots A_n\}$. Then $\Gamma \models B$ if and only if $A_1 \wedge A_2 \wedge \ldots \wedge A_n \models B$

Proof: by def^n of finite set of formulas.

Theorem: $A \models\models B$ if and only if $A \leftrightarrow B$ is a tautology

Proof: This follows from def^n of $A \leftrightarrow B$ & Semantic Deduction Theorem.

Theorem: Let A be a formula. Then A is a tautology if & only if $\neg A$ is NOT satisfiable

special Case

Theorem: Let Γ be a set of formulas and A be a formula. Then $\Gamma \models A$ if and only if $\Gamma \cup \{\neg A\}$ is unsatisfiable.

i.e. Γ = empty

↳ because A is NOT satisfied by what satisfies Γ

Proof: Basically $\Gamma \models A$ means that for every truth assignment ψ that satisfies Γ ha $\psi(A) = T$.

is unsatisfiable.

The cond^n that $\Gamma \cup \{\neg A\}$ is equivalent to the content that for every truth assignment ψ that satisfies Γ has $\psi(\neg A) = F$

Both are thus equivalent.

Truth Tables:

Compact form: It is more of compressed version of the normal truth table.

Let's say we have the formula $(p \to q \to r) \leftrightarrow (p \wedge q \to r)$

Then

p	q	r	$p \to (q \to r)$	$(q \to r)$	$\leftrightarrow$	$(p \wedge q) \to r$
T	T	T	T	T	T	T
T	T	F	F	F	T	F
T	F	T	T	T	T	T
T	F	F	T	T	T	T
F	T	T	T	T	T	T
F	T	F	T	F	T	T
F	F	T	T	T	T	T
F	F	F	T	T	T	T

Reduced Truth Tables:

Insight: if $\varphi(r) = T$ then $\varphi(p)$ & $\varphi(q)$ are not determined.

So. if r is T then $\varphi(p \wedge q \to r)$ is always true!

if r is T then $q \to r$ is always true $\Rightarrow p \to (q \to r)$ is always

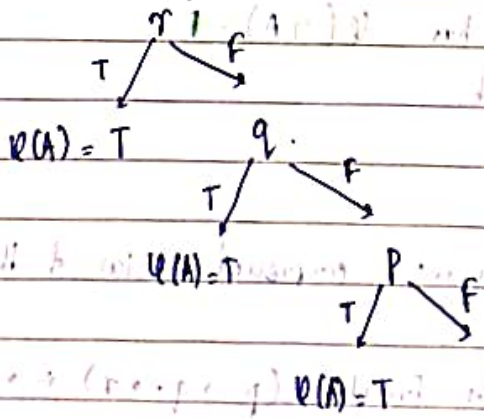
So. we have:

p	q	r	$[p \to (q \to r)]$	$\leftrightarrow$	$[(p \wedge q) \to r]$
-	-	T	T	T	T
-	F	F	T	T	T
T	T	F	F	T	F
F	T	F	T	F	T

Consider this line by line ⇒

Partial truth assignment:

A partial truth assignment means an assignment of truth values to some subset of propositional variables appearing in A.



When a leaf node is reached, the value of the formula is known.

Rules for constructing reduced tables:

(a) Any sub-formula of the form $\neg F$ or $(T \vee B)$ or $(B \vee T)$ or $(T \wedge T)$ or $(F \to B)$ or $(B \to T)$ or $(F \leftrightarrow F)$ or $(T \leftrightarrow T)$ can be determined to have truth value T.

(b) Any sub-formula of the form $\neg T$ or $(F \vee F)$ or $(F \wedge B)$ or $(T \to F)$ or

$(F \leftrightarrow T)$ or $(T \leftrightarrow F)$ can be determined to have both value F.

Examples of some famous tautologies:

1) $p \vee \neg p$ (law of Excluded Middle)
2) $\neg(p \wedge \neg p)$ (Non-Contradiction)
3) $p \rightarrow p$ (Self-Implication)
4) $p \leftrightarrow p$ (Self-Equivalence)
5) $\neg\neg p \leftrightarrow p$ (Double Negation)
6) $(\neg p \rightarrow p) \leftrightarrow p$ (Equivalence with implication) for Negation)

Based Simple Tautological Equivalences:

De-Morgan's law:

$$\neg(p \vee q) \models\models (\neg p \wedge \neg q).$$
$$\neg(p \wedge q) \models\models (\neg p \vee \neg q)$$

Idempotency laws:

$$(p \vee p) \models\models p.$$
$$(p \wedge p) \models\models p.$$

Commutativity laws:

$$p \wedge q \models\models q \wedge p$$
$$p \vee q \models\models q \vee p$$
$$p \leftrightarrow q \models\models q \leftrightarrow p.$$

Associativity laws:

$$p \wedge (q \wedge r) \models\models (p \wedge q) \wedge r$$
$$p \vee (q \vee r) \models\models (p \vee q) \vee r$$
$$p \leftrightarrow (q \leftrightarrow r) \models\models (p \leftrightarrow q) \leftrightarrow r.$$

Distributivity:

$p \wedge (q \vee r) \models\models (p \wedge q) \vee (p \wedge r)$

$p \vee (q \wedge r) \models\models (p \vee q) \wedge (p \vee r)$

Contrapositive :

$p \rightarrow q \models\models \neg q \rightarrow \neg p$

Exportation :

$p \rightarrow (q \rightarrow r) \models\models (p \wedge q) \rightarrow r.$

Other Equivalences :

$(p \vee q) \models\models (\neg p \rightarrow q)$

$(p \wedge q) \models\models \neg(p \rightarrow \neg q)$

$p \leftrightarrow q \models\models (p \rightarrow q) \wedge (q \rightarrow p).$