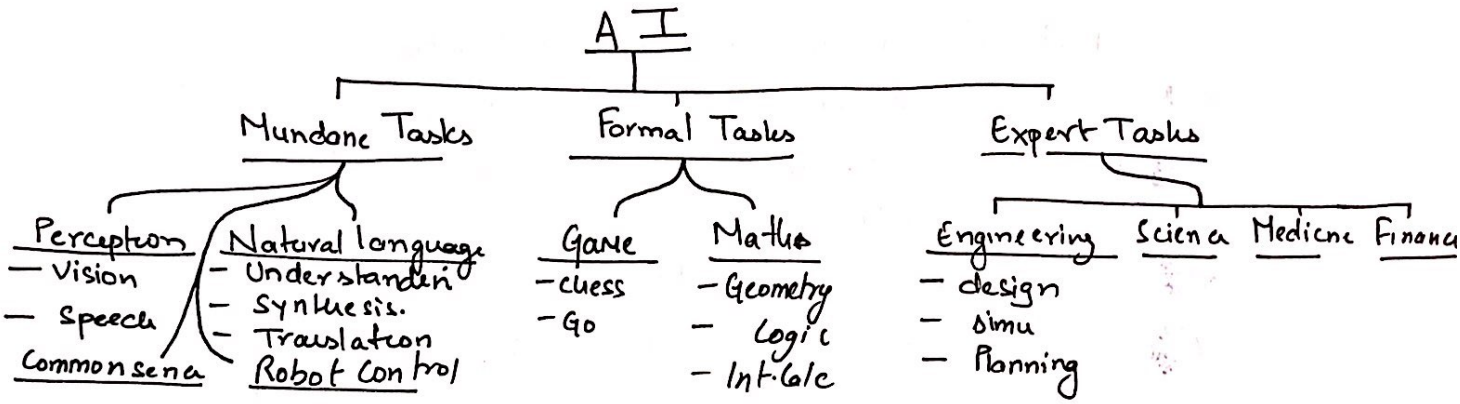
Artificial Intelligence

" AI is the study of how to make computers do things which at the moment, people do better. "

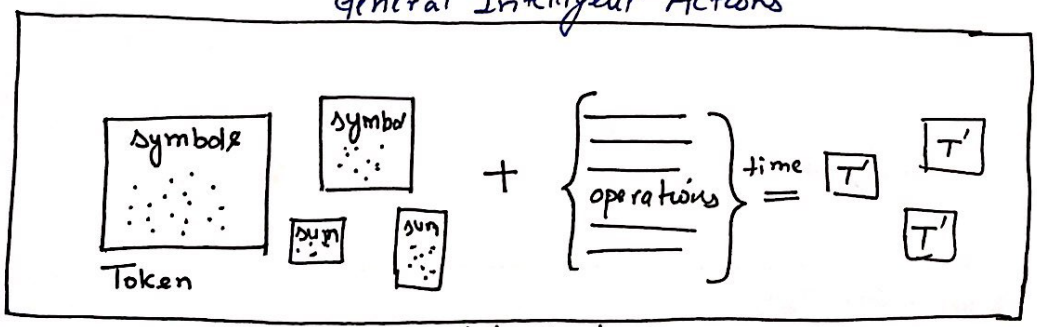


The domains of AI

The Underlying assumption & The AI Technique

> The basic underlying concept is the physical symbol system hypothesis.

" A physical symbol system has the necessary & sufficient means for General Intelligent Actions "



symbol system.

> AI problems are very vast and in an Algorithmic view hard.
i.e No exact answer exists but may approximate.

> It has been shown that any intelligent task requires knowledge. and with knowledge comes drawbacks like.

- Voluminosity
- Low categorization accuracy
- dynamic nature
- differs in organization

} all of these add complexity to the task making the problem Hard.

— The AI Techniques

> It can be concluded that the AI methods exploit this knowledge. knowledge which

一 Capatures generalization, situations that share important properties are grouped together.
二 Is understandable by the people who provide it.
三 Is easily modifiable to correct errors to reflect change in the real world.
四 Is usable even when its not accurate.
五 Can help narrow the range of possibilites.

> It is possible to solve AI problems without the AI Techniques. and it is also possible to solve NON AI problems with the AI techniques.

Problem 1 Tic-Tac-Toe (Brute fora)

- It has 2 DS

1. Board : 9 element grid with elements corresponding to the positions.
   0 means the position is blank.
   1 means X
   2 means O
2. Move table : vector of 3^9 elements with all possible move.

> Play TICTACTOE
> 1. View Table as ( )$_3$ number, convert to decimal ( )$_{10}$
> 2. Index the Htable using the decimal value.
> 3. Set Board eq to the index value.

> Very Time Efficient (Runtime)
> High Memory + Storage needs.
> The Move table is manually filled
> Extending the game is very hard / not possible.

TIC-TAC-TOE (Heuristic)

> It has two DS

1. Board : 9 element vector.
   2 is for blank
   3 is for X
   5 is for O
2. Turn : An Interger showing the turn no.

```
Play TicTacToe

Turn = 1 .   Go(1)

Turn = 2 .   if Board[5] == Blank
                Go[5]
             else Go[1]

Turn = 3     if Board[9] = Blank
                Go[9]
             else Go[3].

Turn = 4     if PosWin(X) != 0
                Go[PosWin(X))
             else go (Make(2)).

Turn = 5     if PosWin(X) != 0
                Go(PosWin(X)) <- Win
             else if PosWin(O) != 0
                Go(PosWin(O))
             else if Board[7] = Blong
                Go(7)
             else go(3).

Turn = 6     if PosWin(O) != 0
                Go(PosWin(O)) <- Win
             else if PosWin(X) != 0
                Go(PosWin(X))
             else Go(Make2)

Turn = 7     if PosWin(X) != 0
                Go(PosWin(X))
             else if PosWin(O) != 0
                Go(PosWin(O))
             else Go(AnyBlank)

Turn = 8     Inverse of T7

Turn = 9     T7
```

```
Make 2 :
   if Board[5] = Blank
      <- 5
   else
      <- BlankNONCorner
```

```
PosWin(p)
  return 0 if P cannot win
  else return the winning
  Position
```

```
Go(n)
  Make a move at
  n if n is valid.
  n=odd => X
  n= even => O.
```

> The winning condition checks each row / col / diag for its product, if its 50 (O) or 18(X) then the corresponding symbol is winning & the blanks addr is returned

> This isn't very efficient due to all the check and multiplications
> This still isn't generalizable to other domains.

> This can be optimized by reordering the values of the board to be a magic square. each player has a set of the positions they've played at and the sum of each pair in the set is used to check for win if the ∘ the difference between 15 and one such sum is ≥ 1 and ≤ 9 then it's a winning pair & the winning position @ the index = difference.

– TicTacToe (Intelligent)

> look ahead at each position that result from each move. and decide on the best rated move.

> This only has one data structure.

BoardPosition: 9 element vector, list of positions possible and an int showing prob of overall win.

> Play TicTacToe
> 1. See if win, if yes give it best Rating.
> 2. Consider all moves op can make next select the worst possible of them & assign it the rating we are considering
> 3. Best move is one with highest rating.

> Requires MUCH more time.

> This generalizes the problem and can be extended to all types of games.

– Summary.

AI Technique

Search
- Provid a way to solve problem with no direct soln along with a frame work to embedd direct techniques

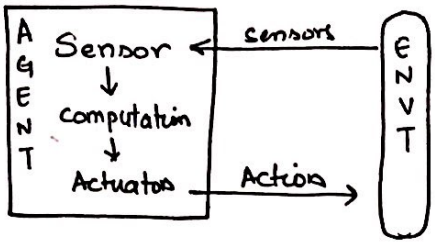
Use of Knowledge
- Provids way to solve complex problem by exploiting the structure of the objects.

Abstraction
- Seperate imp features from noise.

> Programs that use the AI technique are much less fragile, don't get thrown off by small perbutations, people can understand the program knowledge.

Intelligent Agents

An agent is anything that can be percieved to be viewing its envt through sensors and acting upon that envt through Actuators.



Percept Sequence

- It is the complete history of everything the agent has ever percieved. The agents actions @ any time can depend on the entire PS but never on anything it has not seen.

Rational Agent and Rationality -

A rational agent is one that alway does the right thing

Rationality can be defined by:

- The performance measure for success
- Prior knowledge
- The actions agents can perform
- The percept sequence.

Formally, For each percept seq. the RA should select an action that minimizes the error i.e maximize performance given the percept sq & knowledge base.

Rationality ≠ Perfection (deep).

Omniscient Agent

- The agents know the exact actual outcome of their actions and can act accordingly.

Types of Environments

Environment: Anything that is not the agent.

1. Fully / Partially Observable

- Able to perceive everything about the envt vs only some parts

eg) Chess vs Poker.

2. Single vs Multiagent

- No of interacting agent in the observable envt

eg) Chess → Multi agent.
Crossword → Single Agent.

3. Deterministic vs Stochastic

- we can determine what the current action will make us do next time or not

4. Episodic vs Sequential

- Is the agent working only in episodes i.e burst, all data in one ep is lost after it end.
- Is it continiously evolving.

5. Static vs Dynamic

- Does the agent adapt to changes in the envt or is it blind to them.

6. Discrete vs Continious

- Turn based agent interaction vs continious.

Types of Agents

- Simple Reflex : Select actions based on current sq and ignore everthing else.
- Model Based Reflex Agent : Maintains an internal state that depends on the percept history & thereby reflects parts of envt no longer observable.
- ~~Model~~ Goal Based Agent : The agent works based on goal information to decide the next move from current state.
- Utility Based Agents : Goals alone cannot generate high quality actions, some performance is need to check how happy the actions could make the agent.
- Learning Agents : Agents the continiously learn and improve from the envt.

Building Systems To Solve Problems

> To build a system that solves problems we need to do 4 things
  - → Define the problem precisely. This must include the initial, final state.
  - = Analyse the problem. Find the Important features.
  - ≡ Isolate and Represent the task knowledge that is necessary to solve it.
  - 四 Choose the best techniques & solve it.

> One way to model the solution is to consider all possible outcomes. (3^9 in TicTacToe and 10^{120} in chess). This poses some problems.
  1. It is practically impossible to supply all rules correctly
  2. There no efficient way to handle these rules.

> Instead we can define the problem as a set of states where each state is a legal move and a set of rules / production to move between them (control structure).

> The above mentioned has many advantages which are.
  - + Allows for a formal definition as the need to convert some given situation into a desired condition.
  - + Allows to define the solution as a combination of known techniques and Search.

> The first step towards the design of a program to solve a problem must be the creation of a formal & manipulable description of it. This is called operationalization.

Summary -

> To provide a formal desc we must :
  - → Define a state space that contains all the possible configurations of the relevant objects.
  - = Specify one or more states from which the problem may start.
  - ≡ Specify one or more states for which ~~one~~ is the solution.
  - 四 Specify the rules / actions available. considering the assumptions, generalization goal, work requirements.

Production Systems

- As search is the core technique, it is useful to define the task in a way to facilitate it.

- **Production System** :
  - A set of Rules each with a LHS that decides the applicability of the rule and a RHS describing the operation.
  - one/more database of knowledge containing the known information. some parts may be permanent some can change.
  - A control strategy defining the order of application of the rules. and a conflict resolver for multiple matches
  - A rule applier.

Control Strategy -

> A good control strategy causes motion, strategies that do not cause motion will never lead to a solution.

> A good strategy is ~~dso~~ always systemic. This corresponds to the requirement of global motion.

Problem Characteristics

1. Is The Problem Decomposable?

> At each step check whether a problem can be solved immediately or can it be reduced to a form that it knows how to solve.

> Some Problem may not be decomposable through.

Eg> Integration vs Blocks of Words.

2. Can The Steps Be Ignored / Undone?

> Problems can be of 3 types. Ignorable, Recoverable, Irrecoverable.

> The recoverability determines the complexity of the problem.

> Recoverable programs are less complex and can be made using backtracking.

> Irrecoverable tasks are much more complex and may need Planning. Eg> 8 puzzle vs Chess

3. Is The Universe Predictable?

> Is the outcome of actions predictable / deterministic? This again controls the complexity.

> Problems like 8 puzzle are predictable

> Problems like Chess / Bridge / Robot control are non predictive.

4. Is The Solution Absolute or Relative?

> This looks at the any paths vs the best paths problem.

> If the answer is the only thing that's imp and not how we got there then it is a any path problem.

> If the answer is as important as the way we got there then it is a best path problem.

Eg Question Answers vs TSP.

5. Is the solution a State or Path?

> Tasks like languag understanding, the answer is some state of the system. i.e an interpretation.

> Tasks like the water jug problem, we need to maintain the sequence of operations, if the problem is remould to be a state problem, the states we in term become partial paths.

6. What Is The Role of Knowledge?

> Some problems may need basic knowledge to make a move. Adding any more knowledge will only help constraint the moving better eg chess

> Some problem require all the knowledge to reach a solution.
~~these~~ o eg Newspaper scanning.

7. Is The a need to Interact With People?

> Solitary Problems where no communication is needed between the steps.

> Conversational which require comms between steps.

Production System Characteristics

- Monotonic : Application of one rule doesn't make any other rule inapplicable
- NON MONOTONIC : Application of one rule Blocks some Rules.
- Partially Commutative : if a seq of rules changes x to y then any combination of them is allowed.

	Monotonic	Non Monotonic
Partially Commutative	Ignorable Problems Theorem Proving No Back Prop Needed.	recoverable problems Robot Navigation
Non Partially Commutativ	Chemical Synthesis order of op is Imp	Bridge.

Artificial Intelligence Algorithms

1. Generate & Test

- This is the simplest and most ambigious algorithm by definition.
- It has two subprocedures MoveGen and GoalTest.
- MoveGen takes in a state & returns all states that can be reached in one step from it.
- GoalTest Returns True if the state is goal else false.

```
GenerakTest :
    1. while (more children exist)
    2.         Generak a candidate
    3.         Test if it is goal
    4.     return fail
```

- Example MoveGen for River Crossing.

```
MoveGen (n)
    init succ  C = {}
    add M to n to get S
    if Left ∈ n
            S ∪ {RigN}
    else
        S ∪ {left}
    if (legal(S)) add S to C
    for each other E in N
            make S' from S
            S' ∪ {E}
            if legal(S')
            C ∪ {S'}
    return C
```

- Both MoveGen and GoalTest are problem specific.

2. Simple Search

- Set of 3 algorithms that improve in ambiguity, robustness and generalization
- In Simple Search 1 a new Set data structure is used to maintain a list of open states.

```
SS1()
    open ← {start}
    while (!empty(open))
        Pick n from open
        open ← open \ n
        if (GoalTest(n))
            ← n
        open ← open ∪ moveGen(n)
    return failure
```

- SS1 has the possibility of going into ∞ loops in graphs with loops.
- Simple Search2 overcomes this by maintaining a closed list as well.

```
SS2()
    open ← {start}
    closed ← {}
    while (!empty(open))
        pick n from open
        open ← open \ n
        closed ← closed ∪ {n}
        GoalTest(n)
        succ ← moveGen(n)
        open ∪ succ
        open \ closed
    return fail
```

- This overcomes the ∞ loop problem but has the problem of not being able to answer path questions.
- Simple Search3 overcomes this by redefining the ds and introduces 1) MapChar(h) Takes a {{{}}} and returns the head element
  2) Cons(c,n) appends succ to head of node n.

```
BSS()
    open ← {{start}}
    closed ← {}
    while (! empty (open))
        Pick n from open
        h ← head (n)
        if goal Test (h)
            ← Reverse(h)
        else
            succ ← { Move Gen(h) \ close}
            succ ← { succ \ mapchar (open)}
            open ← open \ n
            for each s in succ :
                open ← open ∪ cons (s, n)
    return fail.
```

- This now can return the path which lead to the result.

3. Depth First and Breadth First Search

- This only maintains the parent and child nodes.
- A blind search method.

```
DFS ()
    open ← [[start], [NIL]]
    close ← []
    while ! empty(open)
        npair ← head (open)
        n ← head (npair)
        if GoalTest (n)
            return make_path (npair, close)
        else
            close ← Cons (npair, close)
            ch1 ← More Gen (n)
            nlp ← RemoveSeen (ch1, op, clos)
            news ← MakePair (noloop, n)
            open ← Append (news, Tail (open))
    return fail
```

- Modify the highlighted line by making / swapping the params to get BFS.

Evaluation of Search Algorithms

1. **Completeness** : Does the algo always find the solution if there is one.

2. **Time Complexity** : Unlike Normal T.C measures this uses the number of nodes visited before the solution was found. implementationally this is the No of nodes in closed when the answer is found.

3. **Space Complexity** : Measure of how much space does the algo need to get to the solution. This is equal to the number of node in open when the sol^n is found.

4. **Quality of Solution** : The length of the path from source to solution.

BFS vs DFS (not again)

- **Completeness** : Finite State space both Find Solutions
  Infinite State space DFS may Fail.

- **Time Complexity** :

DFS

at level d with goal at d+1.

$$\left[(d+1) + \frac{(b^{d+1}-1)}{b-1}\right]/2$$

$$= \frac{b^{d+1}}{2b} = O\left[\frac{b^d}{2}\right]$$

BFS

at level d with goal at d+1.

$$\left[\frac{b^d-1}{b-1} + \frac{b^{d+1}-1}{b-1}\right]/2$$

$$= O\left[\frac{b^d(b+1)}{2}\right]$$

$$BFS : DFS = b+1 : b$$

- **Space Complexity**

DFS

$O[(b-1)d+b]$

BFS

$O(b^d)$

- **Quality**
  - Problem dependent

Heuristic / Guided Search

- By adding a heuristic function we can guide the search in the direction of the goal.
- The heuristic value is an estimate of the distance of current from the goal.
- The HV must not be computationally expensive.
- Some Examples of H.V are 1> Euclidean distance 2> Manhattan distance

1> Best First Search

1. This is a modification over DFS where we also add a heuristic value
2. To accomodate this the data structure is modified to

   < parent, child, h value >
3. In the algorithm, the line to update open is also changed.

$$open \leftarrow \underset{h}{Sort}(append(News, Tail(open)))$$

4. The Completeness, Quality and Space depend on the H function.
5. The General approach to select a H function is to minimize Effective Branch and maximize Pentrance.

$$EBF = \frac{\text{Tot No of Nods Exp}}{\text{No of Nods in Sol n}}$$

$$\text{Penetrance} = \frac{\text{No of nods in Sol}}{\text{Tot no of nods}}$$

2> Hill Climbing (Steepest Gradient ascent)

- This is basically a modification of Generate and Test with a h-function
- There is no need of a Goal Test as the algo will always move toward the goal & Halt when it's at a max.
- The algorithm may fail to find a solution.

```
Hill Climbing (S) :
        if S is maxima returns.
        n ← S
        nn ← head (Sort_h (Move Gen (S)))
        while h(nn) < h(n) do:
                n ← nn
                nn ← head (Sort_h (Move Gen (n)))
         return nn.
```

- The major drawback here is the algo can get stuck in
  1) local Max/Min : This point is better than all neighbours but is not the actual best soln.
  2) Plateau : This happens when all values are giving same h value.
  3) Ridge : : A local maxima where along 1 axis the value is const.
- Solution : Backtracking, Momentum, multiple Rules.

3) Depth Bound DFS and Iterative Deepening.

- Adds a max depth level to the DFS algo to prevent infinite searching along a dead branch.

```
Depth Bound DFS (S, Rlimit) :
        open ← ((S, NIL, 0))
        close ← ()
        while !NULL(open)
                np ← head(open)
                n ← head (np)
                if GoalTest (n) :
                        Return Path (np, clos)
                else
                        close ← cons (np, close)
                        if rest( head(np)) < Bound
                                chil ← moveGen(n)
                                nL ← RemoveSeen (chil, open, cl)
                                new ← Makepair (nL, n, Rest (CHC np)))
                                open ← Append ( new , Tail)
        return "NoPath"
```

```
Depth Bound Itrative Deepening()
        Bound ← 1
        while True :
                Depth BoundDFS (S, Bound)
                Bound ++
```

- The Maindrawback of IDBDFS is that it repeats search over-and-over again at lower levels.

4) A* Search

- A* Builds on both Best FS and Branch and Bound. i.e like BstFS it looks ahead and like B&B it looks Back too.

$$f^*(x) = g^*(x) + h^*(x)$$

The actual cost eg

$g(x) \leq g^*(x)$ is the bestt till now $h(x) \leq h^*(x)$ is the prediction

```
A*Search()
1.  open ← {start}
2.  f(start) ← h(start)
3.  Parent(start) ← NIL
4.  closed = { }
5.  while !empty(open)
6.      remove n from open where f(n) is minimum
7.      close ← close U {n}
8.      goaltest(n) return reconstruct path ())
9.      else
10.         neigh ← moveGen(n)
11.
12.         open ← removeSeen (op, cl, neigh)
13.         for each m ∈ neighbour :
14.             if m ∉ open & m ∉ close : add m to open.
15                  g(m) = g(n) + k(n,m)
                    f(m) = g(m) + h(m)
                    Parent(m) = n.
                elif m ∉ close :
                    if g(n) + k(n,m) < g(m) :
                        g(m) = g(n) + k(n,m)
                        f(m) = g(m) + h(m)
                        Parent(m) ← n.
                elif m ∉ open :
                    if (g(n) + k(n,m) < g(m))
                        g(m) = g(n) + k(m,m)
                        f(m) = g(m) + h(m)
                        Parent(m) ← n
                        Propagate (m)
        return Fail.
```

- The propagate (m) updates the score in previously discovered nodes.

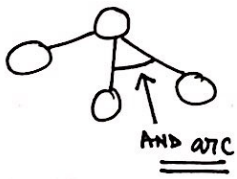
```
Propagate (m):
1.  for each s ∈ More Gem (m):
2.      if (g(s) > g(m) + k(m, s))
3.          g(s) = g(m) + k(m, s)
4.          f(s) = g(s) + h(s)
5.          Parent(s) ← m
6.          if s ∈ close
7.              Propagate (s)
```

- A* will always find a solution when
  1) Branching Factor is Finite.
  2) cost of each more is greater than some ε.
  3) $h(x) \leq h^*(x)$.

5> AO* Search

1> AND-OR graphs

- Used to represent the soln which can be solved by decomposing into smaller problems all of which must be solved.
- This is an "and" arc in the graph.



AND-OR GRAPH.

- The AO* operates on a graph data structure. which represents the search space explicitly generated. each node points to it's successor and predecessor. and a h' value.
- g is not stored, h' serves as the estimate.

AO* Search()

1. G ← start
2. Compute h(start)
3. while start isn't solved or h(start) ≤ Futility
4. Trace the labeled arc from start for exp select one of them as Node
5. Generate successors of Node, if there are none then
6. h(NODE) ← Futility
7.
8. else: for each S ∈ successors
9. Add S to G
10. if S is terminal, mark it solved, h(S) = 0
    else compute h(S).

——— THIS IS THE FORWARD PHASE DONE ———

Propagate the new values up the graph.

1. let N be set of Solved / Modified Nodes.
2. while N is not empty:
3. Select deepest node d & Remove from N.
4. compute best cost (d) from it's children.
5. Mark the best option as Marked
6. if all Nodes connected through Marked are solved
7. label d as solved.
8. if d has changed
9. Add All parents of d to N.

——— THIS IS THE BACK PROPAGATION DONE ———

Finally

1. if start is marked as solved
2. return marked subgraph starting at Start
3. else return failure.

6> Simulated Annealing_

- This is another variation of Hill climbing. but can overcome local optimums.
- Uses Exploitation and Exploration.
- when the algo is stuck at local, this is Exploitation.
- Exploration selected selects some other random local without losing direction

$$P = 1 / 1 + e^{-\frac{\Delta E}{T}}$$

- P is the probability to jump. T is a domain dependant const and ΔE is $h(c) - h(n)$
- as ΔE increases $P \to 1$ & as Δ decreas $P \to 0$.

```
Simulated Annealing()
1.  node ← start
2.  Best ← node
3.  T ← Large INT
4.  for t = I to Epoch
5.      do while (termination)
6.
7.          ng ← RandomN(node)
8.          ΔE ← Eval(ng) - Eval(node)
9.          if Random(0,1) < 1/1+e^(-ΔE/T)
10.             NNode ← Ng
11.             if Eval(ng) > Eval(Best)
12.                 best ← Ng
13.         T ← cool(T, t).
    return Best
```

7> Branch and Bound

```
Branch AND Bound()
1.  open ← {<Start, NIL, cost(Start)>}
2.  closed ← { }
3.  while open is not empty:
4.      pick cheapest n from open.
5.      if GoalTest(n)
6.          reconstruct(n)
7.      else
8.          succ ← MoveGen(head(n))
9.          for each m ∈ succ
10.             cost(m) ← cost(n) + K(m,n)
11.             add(m, head(n), cost(m) to open
```

8> Constraint Satisfaction

- A search technique that operates in a space of constraint sets.
- The goal is a state that has been constrained "enough".
- First the constraints are discovered and propagate as far as possible in the system.
- If there is no solution, A guess is made and searching is done.
- The algo terminates due to
  1> contradiction
  2> No more changes can be made.
- This is a simple 4 step algorithm.

1. Propagate Available constraints. Set OPEN to set of all obj that must have values assigned to them in a complete solution.
   Then do until inconsistency is found/ open is empty
   a) Slet OB from OPEN. strengthen with all avail/ constraints
   b) If this set is diff from set that was assigned last-time then add OPEN to all objects that share constraints with OB.
   c) Remove OB from OPEN.
2. If unions of constraints discovered defines a sol" quit
3. " " " " " " " " contradiction fail
4. Make a guess at something and then continue.

Game Playing

- **Assumptions**
  - We are limited to 2 player board games.
  - Both Players are Rational Players.
  - The game is deterministic and has we defined Rules.
  - The Players play in alternating moves.
  - The games are zero sum games i.e any profit to one player is a loss to the other.

- **GameTree**
  - This is a layered tree with alternating layers showing alt players.
  - One type of player is the Max & the other the Min.
  - The leaves are labeled as win, lose, draw. (with respect to max).
  - The value at root is the minimax value i.e The most probable outcome of a rational game.
  - Max will always maximize its valu & min will make Max Minimum.

→ **The MiniMax Algorithm**

- This is a simple recursive algorithm that defines the minimax procedure.
- It has the following sub modules.

1) Player (S) // defines Player at State
2) Action (S) // possible moves at S
3) result(S,a) // the state achieved by action a at S
4) terminal (S) // checks if S is a terminal.
5) Util(S) // returns the value of a leaf.

$$\text{MINIMAX}(S) = \begin{cases} \text{UTIL}(S) & \text{if } S \text{ is terminal} \\ \max_{a \in \text{ACT}(S)} \text{MINIMAX}(\text{Result}(S,a)) & \text{MAX} \\ \min_{a \in \text{ACT}} \text{MINIMA}(\text{Result}(S,a)) & \text{MIN} \end{cases}$$

```
MINIMAX (S)
    V = MAX_VAL (S)
    return act in succ(S) with value V
```

```
MAX_VAL(S)
    if terminal(S)
        return UTIL(S)
    V = -∞
    for a, s in succ(S)
        V = max (V, min_val (s))
    return V
```

```
MIN_VAS (S)
    if terminal (S)
        return UTIL(S)
    V = ∞
    for a, s in succ(S)
        V = min (V, MAX_val(S))
    return V
```

→ Alpha-Beta Pruning

- The searchable nodes grow exponentially in MINIMAX. To lessen this growth we adopt pruning based on some mathematical constraints.

Eg> MiniMax (R) = max (min (3,12,8), min (2,x,y), min (14,5,1))

= max (3, 2, 2)

= ~~max~~ 3.

now min of 2, x, y will at ~~least~~ most be 2, ∴ x, y = 2, z ≤ 2

Thus there is no need to evaluate x, y.

- To do this we introduce 2 values

  α: The best value we have found any wher for Max

  β: The best value we have found anywher for MIN.

- Each state maintains it own α, β values which get upgraded.

```
αβ Search (St):
    V ← MAX_VAL (St, -∞, +∞)
    return act in Succ (St) with value V.
```

```
MAX_VAL (St, α, β):
    if terminal (St)
        return UTIL (ST)
    V ← -∞
    for a, s in Succ (S):
        V ← MAX (V, MIN_VAL (s) α, β))
        if V ≥ β
            return V
        α ← Max (α, V)
    return V
```

```
MIN_VAL (St, α, β):
    if turminal (St)
        retun UTIL (ST)
    V ← +∞
    for a, s in Succ (St):
        V ← MIN (V, MAX_VAL (St, α, β))
        if V ≤ α
            rctun V
        β ← MIN (β, V)
    return V
```

Predicate Logic

- "Logical formalism is appealing because it immediately suggests a way to derive new knowledge from old, i.e through mathematical deduction."

- **Propositional Logic**
  - Simple
  - Has a decision procedure
  - Represented as well-formed formulas
  - There is no class based coupling i.e No common ID for same types.
  - Cannot cover quantifications.

- **First order Predicate logic**
  - More powerful way of representation.
  - Represented as statements in wff.
  - There is no decision procs, It is only semi decidable.
  - Represented with and-or graphs in statespaces.

- **Computable functions and Predicates**
  - These generalise conditional, computable statements like lt and gt relations.
  - When these are encountered, instead of searching or trying to deduce an answer, we simply invoke a procedure which will simply evaluate the truthness of the arguments.

  eg "No human lives longer than 150 years".

$$\forall x : \forall t_1 : \forall t_2 : human(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2$$

- **Resolution**
  - Works on clauses i.e a special std form.
  - Produces proof by refutation i.e to prove a statement it attempts to show that the negation of it produces an contradiction.

- Conversion of predicate into clause form
- The conjunctive Normal form.
- There is a 9 step algorithm to do this.

Stmut: $\forall x\,[B(x) \rightarrow (\exists y\,[Q(x,y) \wedge \neg P(y)] \wedge [\exists y\,[Q[x,y] \wedge Q[y,x]$

$\wedge\ \forall y\,[\neg B(y) \rightarrow \exists \neg E(x,y)]$

Algorithm

1. Eliminate implications

$a \rightarrow b$ becomes $\neg a \vee b$

$a \leftrightarrow b$ becomes $a \rightarrow b \wedge b \rightarrow a$

Eg> $\forall x\,[\neg B(x) \vee (\exists y\,[Q(x,y) \wedge \neg P(y)] \wedge [\exists y\,[Q(x,y) \wedge Q(y,x]$

$\wedge\ \forall y\,[\neg (\neg B(y)) \vee \neg E(x,y)]]$

2. Reduce scope to negation to atomics.
   1. Demorgans law
   2. $\neg \forall x\,(P(x))$ becomes $\exists x\,(\neg P(x))$
   3. $\neg \exists x\,(P(x))$ becomes $\forall x\,(\neg P(x))$

Eg> $\forall x\,[\neg B(x) \vee [\exists y\,[Q(x,y) \wedge \neg P(y)] \wedge \forall y\,(\neg Q(x,y) \vee \neg Q(y,x)] \wedge \forall y\,[B(y) \vee \neg E(x,y)])]$

3. Standardize variables

$\forall x\,(P(x) \vee \forall x\,Q(x)$ becomes $\forall x\,P(x) \vee \forall y\,(P(y))$.

$\forall x\,[\neg B(x) \vee [\exists y\,(Q(x,y) \wedge \neg P(y)] \wedge \forall y\,(\neg Q(x,y) \vee \neg Q(y,x)] \wedge \forall z\,[B(z) \vee \neg E(x,z)])]$

4. Move all quantifiers to the left.

$\forall x\,\forall y\,\forall z\,[\neg B(x) \vee [\exists y\,(Q(x,y) \wedge \neg P(y)] \wedge [\neg Q(x,y) \vee \neg Q(y,x) \wedge [B(z) \vee \neg E(x,z)])]$

5. Eliminate the existential quatifier with the Skolem.

$\exists y : P(y)$ becomes $P(S_1)$.

$\forall x\,\forall y\,\forall z\,[\neg B(x) \vee [Q(x,S_1) \wedge \neg P(S_1)] \wedge [\neg Q(x,y) \vee \neg Q(y,z)] \wedge [B(z) \vee \neg E(x,z)])]$

6. Drop the prefixes.

$$[\neg B(x) \vee ([Q(x,S_1) \wedge \neg P(S_1)] \wedge [\neg Q(x,y) \vee \neg Q(y,x)] \wedge [B(z) \vee \neg E(x,z)])]$$

7. convert the sentence into a conjuncture of disjuncts.

$E_1 \vee [E_2 \wedge E_3 \ldots E_n]$ becomes $[E_1 \vee E_2] \wedge [E_1 \vee E_3] \wedge \ldots [E_1 \vee E_n]$.

$E_1 \wedge [E_2 \vee E_3 \ldots E_n]$ becomes $[E_1 \wedge E_2] \vee [E_2 \wedge E_3] \ldots [E_1 \wedge E_n]$.

Eg>

$$[(\neg B(x) \vee (Q(x,S_1) \wedge (B(x) \vee \neg P(S_1))) \wedge (\neg B(x) \vee \neg Q(x,y) \vee \neg Q(y,x)) \wedge \neg B(x) \vee B(z) \vee \neg E(x,z))]$$

8 Create a seperate clause for each conject

$$[\neg B(x) \vee (Q(x,S_1)]$$
$$[\neg B(x) \vee \neg P(S_1))]$$
$$[\neg B(x) \vee \neg Q(x,y) \vee \neg Q(y,x)]$$
$$[\neg B(x) \vee B(z) \vee \neg E(x,z)]$$

9. ~~Purge~~ Rename all variables

$$[\neg B(x) \vee (Q(x,S_1)$$
$$\neg B(w) \vee \neg P(S_2))$$
$$\neg B(u) \vee \neg Q(u,v) \vee \neg Q(v,u)$$
$$\neg B(a) \vee B(z) \vee \neg E(a,z))$$

Better algo

1. Eliminate $\rightarrow, \leftrightarrow$
2. make $\neg$ atomic
3. Purge $\exists$ with skolem
4. Rename variables
5. move quants left
6. move disjunctures docon
7. Eliminate conjunctions
8. Renamevariables
9. Purge quantifiers.

The Resolution Process

- It is a simple Herative process ; at each step two parent clause are compared and resolved yielding a new infered clause.
- If an empty clause is produced, then a contradiction has been found.
- If a contradiction exists it will be found, but if there is not contradiction then -the algo may go ran forever.

- **Proposition Resolution**

Algorithm

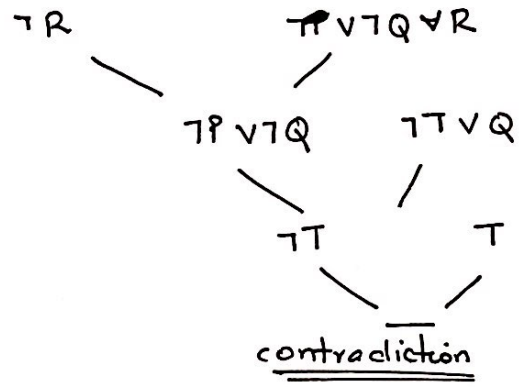
1. Convert all propositions to clause form.
2. Negate P and add it to the clauses obtained in 1.
3. Repeat till a contradiction is found or no progress can be made.
   a) Select two clauses, call them parent clauses.
   b) Resolve them, the resultant is a clause called resolvent. It is the disjunction of all of the literals of both parents with L and $\neg L$ eliminated.
   c) If the resolvant is empty, a contradiction has been found. if not add it the set of clauses.

E.g.

1. P
2. $\neg P \vee \neg Q \vee R$
3. $\neg S \vee Q$
4. $\neg T \vee Q$
5. T

To prove R add

1. add $\neg R$ to set



The unification algorithm

- With predicates, the matching process becomes complicated, this is because not only the predicate but the arguments must also be considered.
- To find contradictions, the process much compare two literals and discover if there a set of substitutions that makes them same/identical.
- If the predicate symbols match, we check the arguments one pair at a time, if the first match we can move onto the next.
- To test each pair we call the algo recursively
  - **Rules** 1. different predicates and consts cannot match
    2. A variable can matching any var, const, pred as long as the pred doesn't have an instance of that variable.

– Substitutions

- A substitution is shown as y/x or substitute x with y.
- when a sub is done, it is distributed over all literals i.e

$$C\,a_1/a_2, a_3/a_4 \ldots)(b_1/b_2, b_3/b_4)$$

means first do the sub in Right most list then take the results and apply all the ones of next list.

Algorithm

Unify (L_1, L_2):

1. if L_1 and L_2 are both vars or consts then :
   a) if L_1 and L_2 are identical : return NIL.
   b) else if L_1 is var then if L_1 occurs in L_2 return Fail else return (L_2/L_1)
   c) else if L_2 is var then if L_2 occurs in L_1 return Fail else return (L_1/L_2)
   d elsse return fail
2. if the predicate symbols in L_1 and L_2 are not same then return Fail
3. if L_1 and L_2 have diff no of args then return fail.
4. Set SUBST to NIL
5. for i ← 1 to no of args in L_1 :
   1. call unify with i^{th} arg of L_1 with i^{th} arg of L_2 ; put result in Subst.
   2. if S contains fail return fail.
   3. if S != NIL :
      a) Apply S to remainder of L_1 and L_2
      b) Subst := append (S, SUBSTR)
6. Return SUBSTR.

Resolution of Predicate Logic

- Two literals are contrary if they can be unified with negation of the other.
- The unification algo is vital in this.

Algorithm

Resolution

1. Convert all statements of F into CNF.
2. Negate Paul convert to CNF, add this to the set from 1.
3. Repeat until a contradiction / no progress / predetermined exit crit
   1. Select two clause, call these parent.
   2. Resolve them. The resolvant will be the disjunction of all the literals of both parent clauses with appropriate substitution performed.
   3. if resolvant is empty, we have a contradiction if not add it to the set of clauses and continue.

Strategy for selecting clauses

- only resolve clauses with complementary literals.
- eliminate tautologies (~~that~~ can never be unsatisfiable) and clause that can be subsumed.
- Try to resolve with clause which has the stmnt which we are trying to refute. (set-of-support strategy)
- Try to resolve with unit / single literals. (unit-preference strategy).

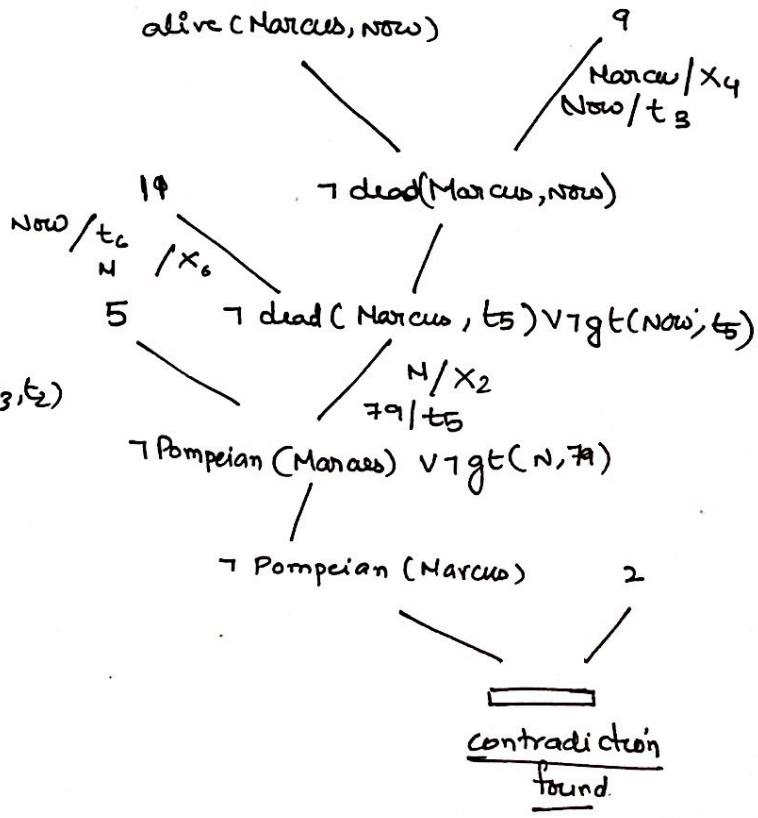
Eg

F:

1. man (marcus)
2. Pompeian (marcus)
3. born (Marcus, 40)
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg pompeian(x_2) \vee died(x_2, 79)$
6. erupted (vocahot, 79)
7. $\neg mortal(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. now = 2008
9. $\neg alive(x_4, t_3) \vee dead(x_4, t_3)$
10. $dead(x_5, t_4) \vee alive(x_5, t_4)$
11. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

P: $\neg alive(Marcus, now)$

$\neg P$ = alive (Marcus, now)

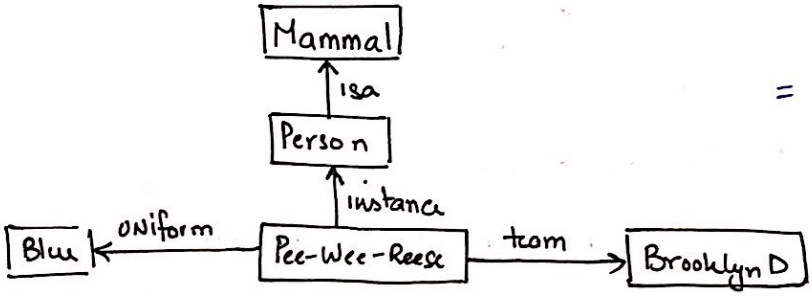


Slot and Filler Structures

- Knowledge is structured as a set of entities and their attributes.
- Make inheritance of knowledge very easy.
- Indexes assertions by the entities they describe.
- Makes it easy to describe properties of relations.
- Has all advantages of OOP.

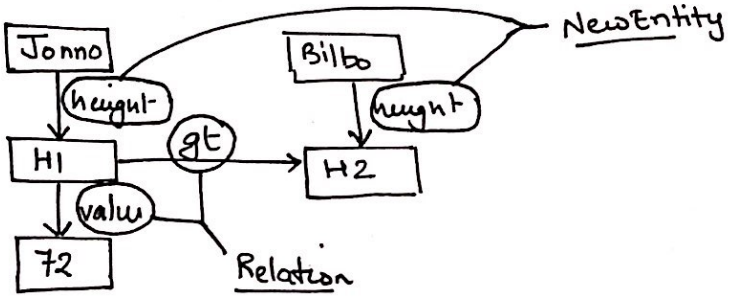
Semantic Nets

- The meaning of a concept comes from the way in which it is connected to other components.
- Information is represented as a set of nodes.
- The connection between nodes represent the relationship.



= { isa(Person, Mammal)
instance(PWR, Person)
team(PWR, BD)
UniCol(PWR, Blue)
}.

- Many unary predicates can be represented as binary predicates using general purpose predicates like instance and isa.
- 3or more place predicates can be converted by creating one new object representing the entire statement and then introducing bin to describe it.
- There are two type of links, one that define new entities and ones the difers the relation between two entities.



- Each node also has its own properties and the ones it inherits / passes on.

Intersection Search

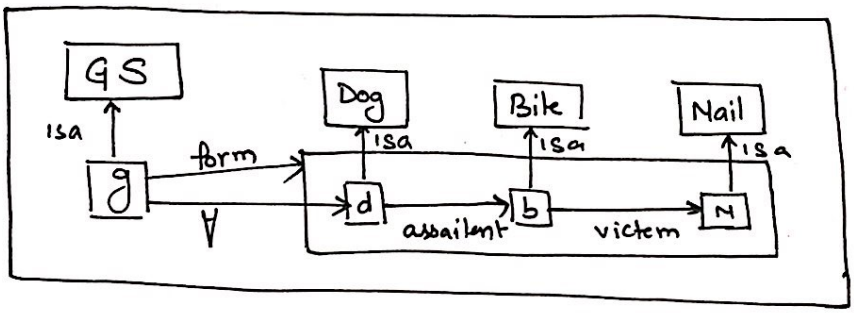
- a.ka.: Marker Passing, Activation Propagation, Spreading Activation.
- Used to find Relation b/w objects by spreading activation out from each of the two nodes and seeing where they meet.
- This exploits the entity based organization of knowledge.

Partitioned Semantic Nets

- Partition the semantic net into a hierarchical set of spaces, each of which corresponds to the scope of one or more variables.
- There is an instance of a class GS i.e general statement for universally quatified variables
- Every GS has two attributes, a form and one or more $\forall$ connections.
- The spaces of a semantic net are partitioned by an inclusion hierarchy.

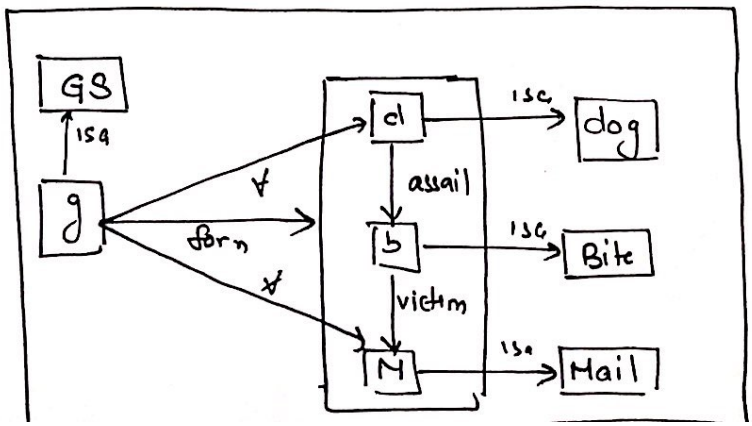
" Every dog has bitten a mail carrier ".

i.e $\forall x : Dog(x) \rightarrow \exists y : MailCarrier(y) \wedge Bite(x, y)$



" Every dog has bitten every mail carrier "

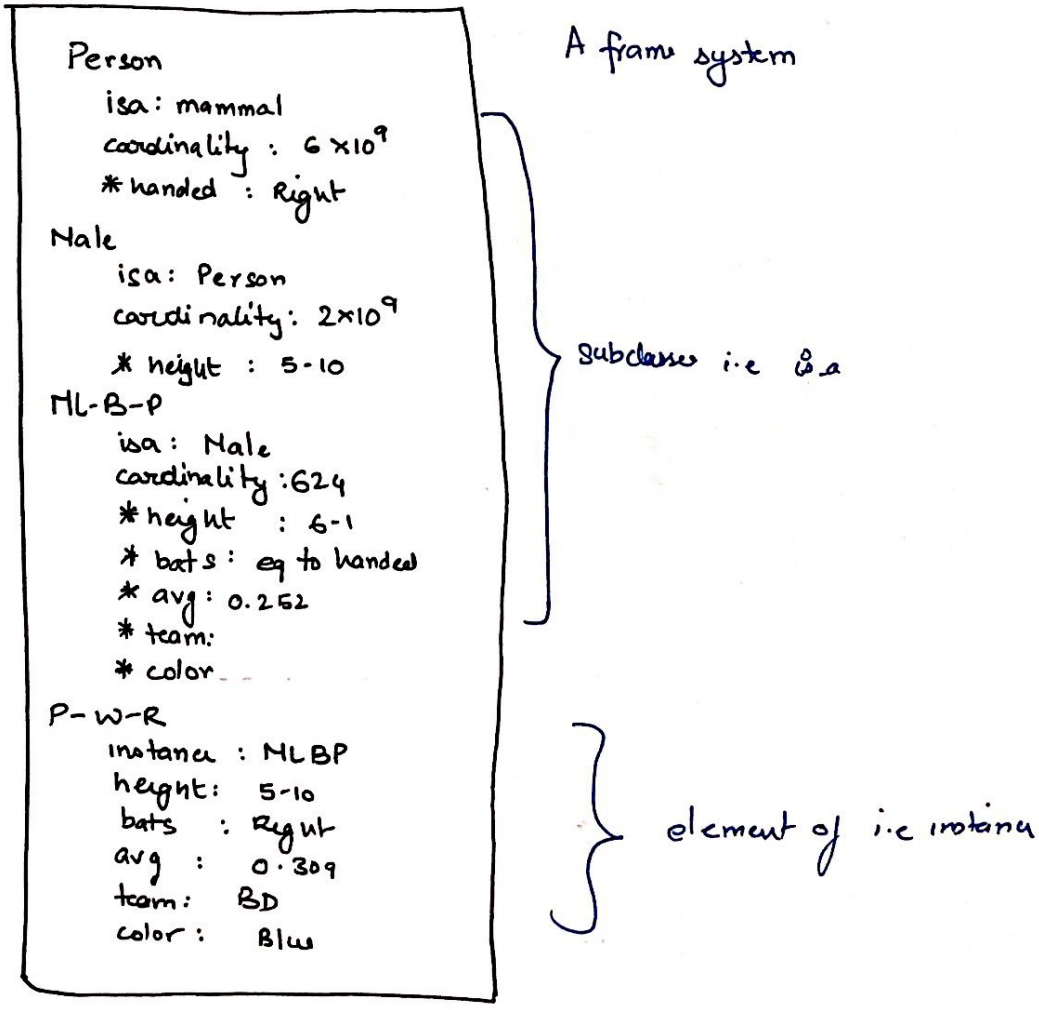
$\forall x \forall y \; Dog(x) \wedge Mail(y) \rightarrow Bite(x, y)$



Frames

- A frame is a collection of attributes (usually called slots) and associated values that discribe some entity in the world.
- The representation can be absolute or relative.
- A single frame when alone is rarely useful. We use a system of frames.
- A value in one frame may be a frame itself.

- Each frame represents a class or an instance.
- isa clause is the subset relation
- element of clause is the instance relation.
- Both isa and instance have inverse attributes, i.e subclasses and all instances.
- There are two kind of attributes; the ones about the set itself and the ones which are to be inherited.
- The inheritable values are marked with an *.
- The isa clause can have if added and if needed procedures.



- Classes are entities themselves. They have pvt as well as ptd values.
- Classes must be both an isa of a larger class and an instance of a class of sets.
- We can diredetlee representation as
  - Classes who's values are entities
  - MetaClasses which are special classes with elements as classes.
- A class is now an instance of some while isa of others.

```
class
    instance : class
    isa      : class
    * cardinality

Team
    instance : Class
    isa      : class
    cardinality : {teams}
    * team size :

NLBTeam
    instance : Class
    isa      : Team
    cardinality : 26
    * team size : -
    * manager :

BD
    instance : NLBTeam
    isa      : NLBTeam
    teamsize : 24
    manager  : some guy
    * color  : Blue.

PWR
    instance : BD
    instance : fielder.
```

A frame system using metaclasses.

• Conceptual Dependencies

- Semantic Nets and Frame systems may have specialized inference procedures and links but there are no rules about what kind of objects and links are good of knowledge representation.

- CD is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences.
  - It facilitates drawing inferences from sentences.
  - Independent of language in which the sentences were originally stated.

- The represented with conceptual primitives that can be combined to form meaning of words in any particular language.
- CD provides a structure and a set of primitives at a level of granularity out of which representations can be made.

- Set of Primitives

ATRANS : Transfer of Abstract rel (give)
PTRANS : Tranfer of Physical location (go)
PROPEL : Application of physical force (push)
MOVE : Movement of bodypart by its owner (kick)
GRASP : Grasping of object by owner (clutch)
INGEST : Ingestion of an object by an animal (eat)
MTRANS : Transfer of mental info (tell)
EXPEL : Expulsion of something from the animal body (cry)
MBUILD : Building of new information (decide)
SPEAK : Production of sound (say)
ATTEND : Focusing of a sense organ (listen)

- Primitive Conceptual categories

ACTs : Actions.
PPs : Object (picture producers).
AAs : Modifier of actions (action Aides).
PAs : Modifier of PPs (Picture aides).

Dependency Structure

1. PP ⟺ ACT — John ⟺ (P) PTRANS — John Ran : Relⁿ b/w a ctor and an event they caused.

2. PP ⟺ PA — John ⟺ height (>avg) — John is Tall : Relⁿ b/w a PP and a PA that is being asserted to describe it.

3. PP (≡) PP — John (≡) Doctor — John is a Doc : Relⁿ b/w two PPs one of which belongs to the set defined by other.

4. PA → PP — nice → boy — Anice boy : Relⁿ b/w a PP and an attribute that has already been predicted for it.

5. PP ⇒ PP — John ⟹ (Poss By) dog — John's dog : Relⁿ b/w two PP, where one provides some info about the other, possession / Location / containment.

6. ACT ←(O)— PP — John (⟺)(P) PROPEL ←(O)— cart — John pushed the cart : Relⁿ b/w an ACT and the PP which is the object of this act.

7. ACT ←(O)— PP, PP — John (⟺) ATRANS ←(O)— mary, John; ATRANS ↑ Book — John took from mary a book : Relⁿ b/w an ACT with a source & dest.

8. ACT ←(I)— ⇓ — John ⟺(P) INGEST ←(I)— John; INGEST ↑(O) ICECREAN; John ↑(O) spoon — John ate ice cream with a spoon : Relⁿ b/w an ACT and the instrument with which it is performed.

9. ACT ←(D)— PP, PP — John ⟺(P) PTRANS ←(D)— field, John; PTRANS ↑(O) Fertilizer — John fertilized the field : Relⁿ b/w ACT and its b/w Physical source & dest.

10. PP ⟸ PP, PA — Plants ⟸ Size > x, Size = x — The plants grew : Rel b/w PP between two states.

11. ⟺ OR ⟺ — Bill ⟺ Propel ← Bullet ← Bob, GUN; Bob ⟸ health (-10) — Bill shot Bob : Relⁿ b/w one conceptualization and the other that causes it.

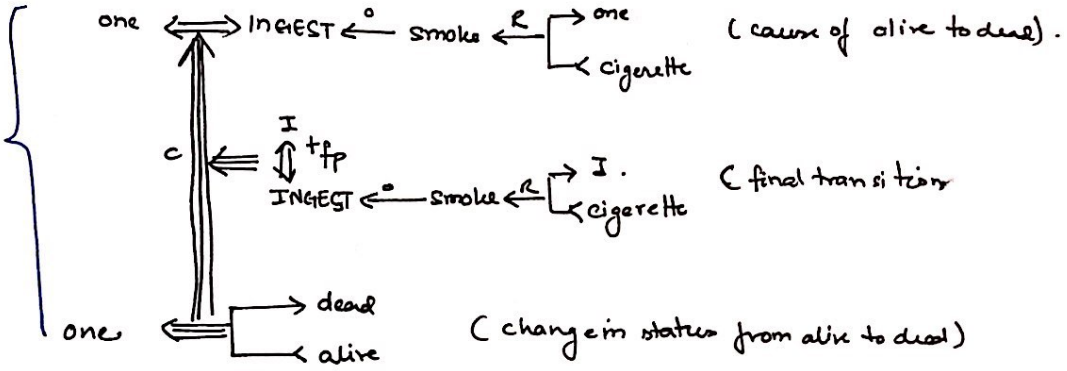
12. I ⟺ — John ⟺(Yest I) PTRANS — John Ran yesterday : Relⁿ b/w a conceptulizeration and time.

13. ⟺ ↓ ⟺ — I ⟺ PTRN ←(O) I ←(D) home, I; I ⟺ NTRA ← frog ←(E) CP, eyes — while going home I saw a frog : Relⁿ b/w a conceptulization and another which is the cause of it happening.

14. PP ⇓ ⟺ — woods ⇓ — I ⟺ NTRANS ← frog ←(E) CP, ears — I heard the frog in the woods : Relⁿ b/w a conceptulization and the place where it happened.

- There are 3 main benifits of using CDs
  1. Fewer inferena rules are needed
  2. The Representation of self contains them inferences
  3. The initial Representation might have holes which can be the point of attention later.

using modifier to adde more meaning to the CD.



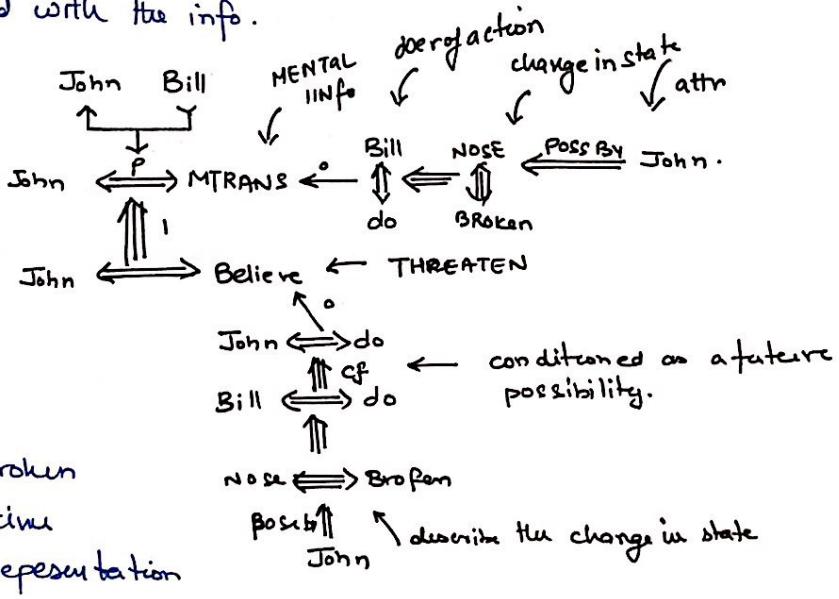
Advantages

- 1 implies that we only need the rules for the primitive ACTs. In a CD these can be stated once and associated with the ACT

  eg give / Take / steal / Donate are of type ATRANS.

- 2 imples to make a CD we must not only use the info in the sentence but also the set of rules associated with the info.

Eg>



- 3 Unspecified info can be point of focus as in when it is encountered.

Drawbacks

1> Required everything be broken down into a set of primitives this causes some trivial representation to become very large.

2> It can only describe events.

Scripts

- A script is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots.
- Each slot may be some information about what kind of values it may contain as well as a default value to be used if no other info.
- Scripts are useful because in the real world there are patterns in the occurance of events which are due to the causal relation b/w events. i.e causal-chain.
- Within a causal chain, the events are linked to the ones that caused them and the ones they enable.
- If the script is known to be appropriate for a given situation then it can be very useful in predicting occurance of events that were not explicitly mentioned.

- Components of a script

Entry Condition	: The condition which in general must be true for the script to occur.
Result	: The condition which in general will be true after the script occurs.
Props	: Slots representing objects involved in the events of the script. These can be inferred
Roles	: Slots representing people involved in the scripts. This can be inferred if not mentioned.
Track	: The specific variation / more general path represented by the script.
Scenes	: The actual sequence of events that occur.

Triggering A Script

- For fleeting scripts, it may be sufficient to merely store a pointer to a script so that it can be accessed later if necessary.
- For non fleeting scripts, it is appropriate the script fully and to attempt to fill in its slots with particular objects / people involved.

1- Once a script is activated there are a variety of ways in which it can be useful in interpreting a situation.

2- Scripts provide a single coherent interpretation from a collection of observations.

3- Scripts allow for a way to focus attention on unusual events.

Natural Language Processing-

- Focuses on spoken language.
- Needs facilities for both written language understanding as well as enough knowledge to handle all the noise and ambiguities of audio signals.

Language Processing-

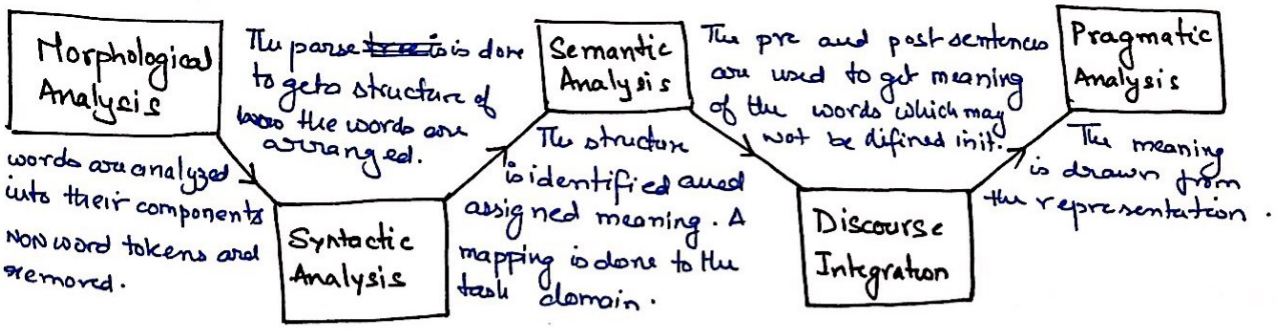
- Processing Written text using Lexical, syntactic and semantic knowledge of the language as well as the real world info.
- Processing spoken language using all info from written as well as additional knowledge about phonology and the ambiguities of speech.

- NLP includes both Understanding and Generation of language and other tasks like translation.

- Language can be difined as a set of strings without reference to any word being describe or task being performed.
- Language is a pair (source and target representation) together with a mapping between the elements.

- Steps in NLP

- The process of NLP can be split in 5 processes with boundaries which may be at times fuzzy.



- The phases may be done sequentially or all at once.
- The phases may be done seperatly but the way that they operate it is nearly impossible to seperate them completly.

Problem Decomposition

- Processes and knowledge Required to do the task
- The global control structure imposed on the process

Morphological Analysis

- Removes non word tokens like punctuations, white spaces etc
- Separates out the word and any prefix/suffix like possesives.
- assigns syntactic categories to all words in the sentence. to help with the interpretation of affixes.

Syntactic Analysis

- Uses the result of the M-A to build a structural description of the sentence. a.k.a parsing.
- Transforms a flat list into a structure that defines the units that are represented in it.
- The structure is designed to correspond to the sentences units that will correspond to the meaning units.
- Creates a set of reference markers which correspond to some entity mentioned in the sentence.
- The markers provide a place to accumulate information about the entities

Semantic Analysis

- Maps the individual words to an appropriate object in the knowledge base.
- It must create the correct structures to correspond to the way the meaning of the individual words combine with each other.
- Depending on the knowledge base we can make a partial meaning using the referen markers.
- Eg Frame Systems.

Some processes maybe collapsed into one if the task in hand allows for it.

Discourse Integration

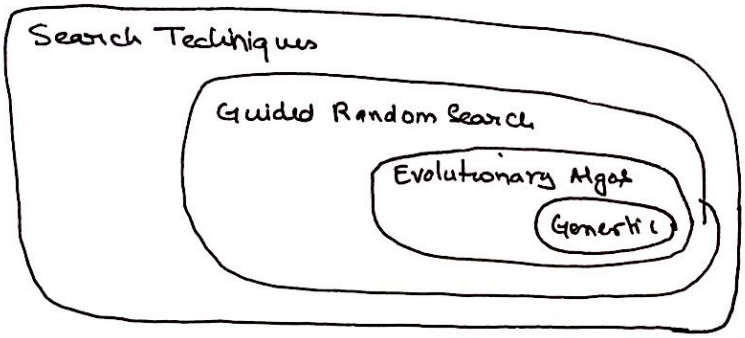
- Identify which individuals and objects are being refered to.
- A model of current discourse context from which we can learn the current user.
- A model to identify the object.

Pragmatic Analysis

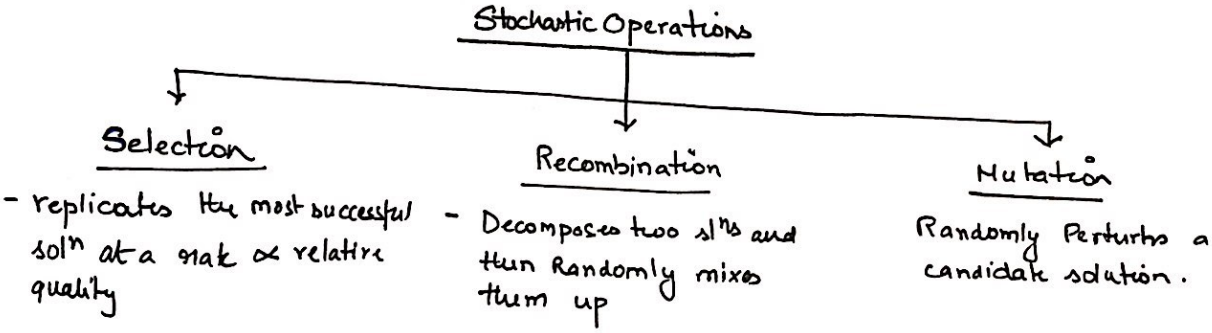
- decide what to do as a result,
- can be to :
  - record what is said and be done with it.
  - Try and discover ~~the~~ intended effect using rules that characterize cooperative dialogues.

Genetic Algorithms

- Probabilistic optimization algorithm which uses natural selection and Genetic Inheritance (Holland 1975)
- Well suited for hard problems with little known knowledge of search space.



- GA maintain a population of candidate solutions and makes it evolve by applying a set of stochastic operations.



Intuition

optimization :: Envt ; FeasibleSoln :: Peoplin Envt ; Soln Quality :: degree of adapt

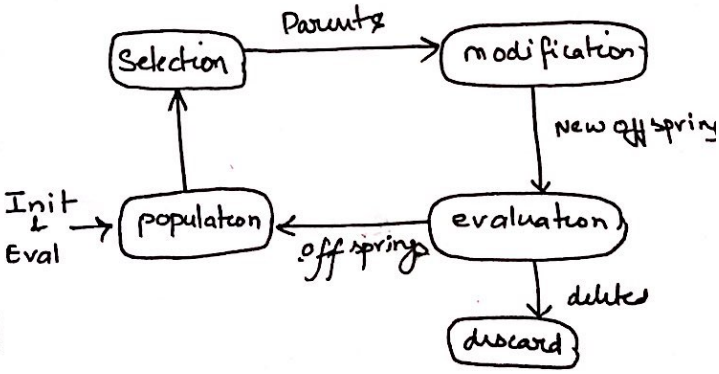
Set of Soln :: population ; operators :: NaturalSel ; iterative enhanc :: evolution.

It can produce complex structure of solutions while staying a simple process.

BluePrint of GA

1. Produce population of soln
2. evaluate quality of all individuals
3. While terminated :
4. Select better solns
5. Recombine
6. mutate
7. evaluate quality
8. generate new population

Cycle of evolution



The MaxOne Problem

- To maximize the number of 1 in a string of size L.
- This is an extreme simplification of maximizing number of correct ans.

features:

+ An individual is a string of size l
+ the Fitness of an individual is no of ones.
+ The start is a population of n random strings.

Eg L= 有8 n=6

we have

S_1 :	1111 0101	f = 6
S_2 :	0111 0001	f = 4
S_3 :	1110 1100	f = 5
S_4 :	01000100	f = 2
S_5 :	11101111	f = 7
S_6 :	0100 1100	f = 3

} initialization

Selection any individual i has the following prob of being selected.

$$p(s_i) = \frac{f(i)}{\sum_k f(k)}$$

S_1' =	11101111	(S_5)
S_2' =	1111 0101	(S_1)
S_3' =	1110 1100	(S_3)
S_4' =	0111 0001	(S_2)
S_5' =	0100 1100	(S_6)
S_6' =	11101111	(S_5)

Crossover take any two strings and select a rand crossover point ; crossover is done at 60 % prob of happening.

Eg> S_1', S_2' ; point = 4

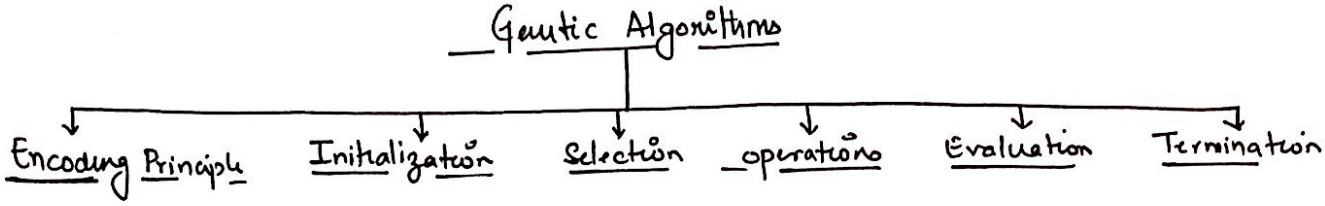
⇒ S_1'' = 1111 1111

S_2'' = 1110 0101

Mutation : at ~~0.1~~ ~~1~~ 10 % prob apply a mutation to any random bit the each string ~~...~~ and if it is true invert it.

- Recalculate The fitness scores.

Components of GAs



1. Encoding / Representation

They can be : Bitstrings, Real numbers, Element Permutation, Rule List etc any data structure is permissible.

Guidelines for selecting a DS

1. Stay as close as possible to natural representation.
2. write operators wrt DS.
3. if possible ensure all genotypes correspond to soln.
4. if possibe ensure operator ensure feasibility.

2. Initialization

1. Start with a population of randomly generated individuals
2. USE a previously saved population.
3. Solutions provided by human expert.
4. solutions provided by some other algorithm.

3. Selection

- survival of the fittest (literally)

A) Fitness Proportionate Selection

- optimal trade off b/w explore & exploit

Drawbacks :

- different selection for $f_1(x)$ and $f_2(x) = f_1(x) + c$.
- Superindividuals may cause premature convergence.

B Linear Ranking Selection

- Based on sorting individuals on fitness.

$$p(i) = \frac{1}{n}\left[\beta - 2(\beta-1)\frac{i-1}{n-1}\right] \; ; \beta \in [1,2]$$

β is the sampling rate of best individual

c) Local Tournament Selection

- Extract k individuals from population without reinsersion and make them play a game where the prob of winning is $\propto$ to the fitness
- The selection pressure is directly $\propto$ to k.

4. Recombination

- Enable movement towards more promising regions of search space.
- Matches good parent subsolutions to make better off springs.

5. Mutation

Simulates the low probability mutation / error that happen during duplication.

- causes movement in search space.
- Restores lost info into the population.

6. Evaluation

- Determines quality of solution ; choosing a good eval is hard.
- Similar solutions should have similar fitness.

7. Termination Condition

- Fixed number of iterations.
- Set quality threshold achieved.
- No improvement / plateaued values.

Why Do GAs Work

$\{0, 1, \#\}$ ⟶ Alphabet

Schema: A template containing a string composed of the alphabet.

Order: No of no wild card positions on the schema.

There are $2^{L-O(S)}$ strings matching S.

O(S) helps decide survival prob

defining Length: $(\delta(S))$ distance between first and last fixed char.

$\delta(S)$ decides prob of crossover.

$m(S,t)$: No of individuals belonging to S at time t.

$f_s(t)$: average fitness of S at t.

$f(t)$: average fitness of overall strings.

Selection

Expected No. of individual of S at time $t+1$ is

$$m(S, t+1) = m(S,t) \cdot f_s(t) / f(t)$$

if S remains above average by const c ; $c \geq 0$

$$m(S,t+1) = m(S,t)[1+c]^t$$

⇒ Above Avg schemas get exponentially increasing strings in next gen.

Crossover

Probability for S of size L to survive crossover is

$$P_s(S) \geq 1 - P_c(\delta(S)/L-1)$$

⇒ above Avg schemas with short $\delta(S)$ will be sampled at exponential increase

$$M(S,t+1) \geq m(S,t)\left[f_S(t)/f(t)\right]\left[1 - P_c(\delta(S)/(L-1)\right]$$

Mutation

Prob of S to survive mutation is

$$P_s(S) = (1-P_m)^{o(S)}$$

$$[P_m \ll 1] \therefore \quad P_s(S) \approx 1 - P_m \cdot o(S)$$

Thus we have:

$$M[S,t+1] \geq m(S,t)\left(f_s(t)/f(t)\right)\left[1 - P_c\left(\delta(S)/L-1\right) - P_m \cdot o(S)\right]$$

Schema Theorem

Short, low-order, above average schemas recieve exponentially increasing trials in subsequent generations.

i.e. GA explore search space by short, low-order schemata.

This is also called the Building Block Hypo.

- It applies to many cases but is very dependent on representation.

Eg of Exception

$S_1 = [11\ \#\#\#\#\#\#]$ $S_2 = [\#\#\#\#\#\#11]$

will give

S_3 [111 ##### 11] and S_4 [000 ##### 00]

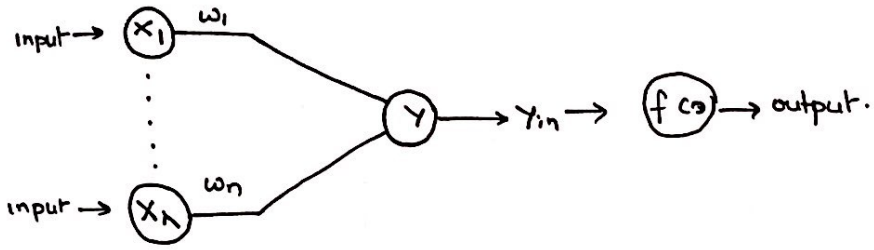
$$f(S_3) \leq f(S_4)$$

GA tend to get misguided and converge to local optimums.

Solution: Better fitness functions or uses the **inversion operator**

Artificial Neural Networks

- Info processing system designed to mimic the biological neuron.
- Applications : Classification, clustering, pattern Recognition etc.



Neuron output = $$Y_{out} = f\left(\sum_{i=1}^{N} x_i w_i\right)$$

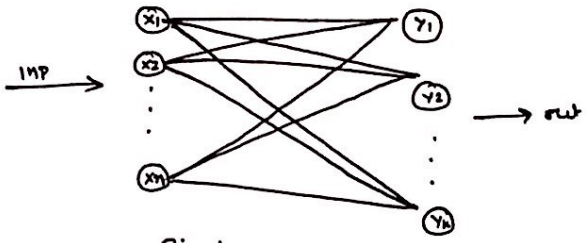
- w are the 'weights' of an input node i.e How much they contribute to the soln
- f() is the activation / thresholding function which 'Activates' when the y value crosses some threshold.

ANN vs NNN

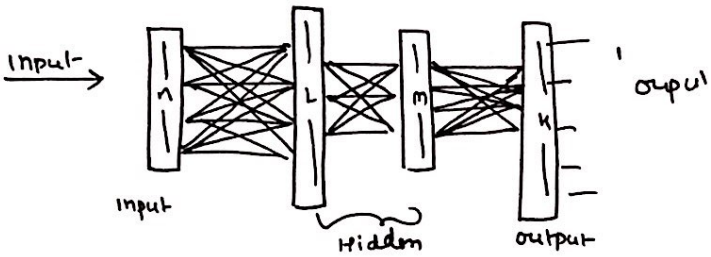
ANN		NNN
- Nano secs	>	milisecs
- Processing: Serial	≤	Massivly Parallel
- Restricted to Complimits	<<	very big 10^{15} Neurons.
- Memory may be restricted but is forever accessible	≈	The brain can forget where some data is
- No fault tolerance	<<	Fault tolerant
- Very simple control Mech	<<	Very complex control

- Neurally implemented math model
- Large no of highly connected neurons.
- Weighted links
- Can Learn, recall & generalize
- The info is represented by allection of neuron.

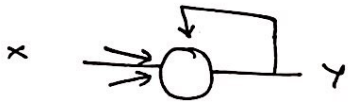
Layouts



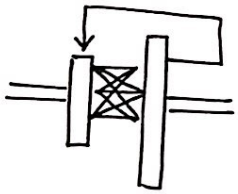
Single Layer FF.



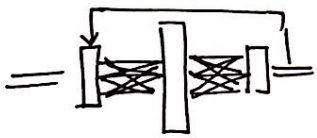
Mullifay FF



Single Node Feedback.



Singlelayer Recurrent



Mullilayer Recurrent

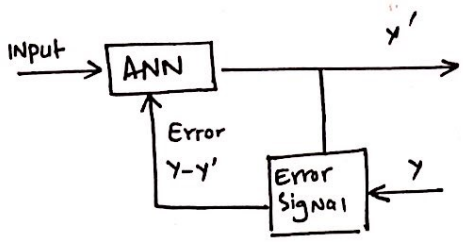
Learning :

1> Parameter Learning : update the wts.

2> Structure Learning : change the structure.

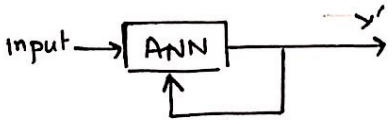
- A more finer way to classify learning is :

1. **Supervised** : output is known. i.e each data item has input and target ~~out~~ o/p.



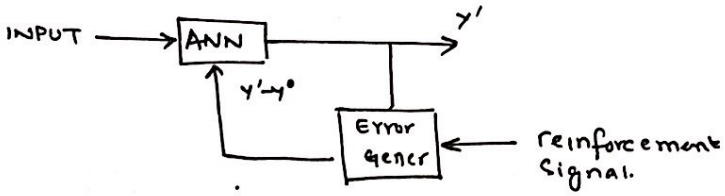
Supervised

2. **Unsupervised** : There is no defined target output. They work by grouping similar items together



unsupervised

3. **Reinforcement** : Similar to supervised but instead of target output we get a critic info which tells how good the ann is.



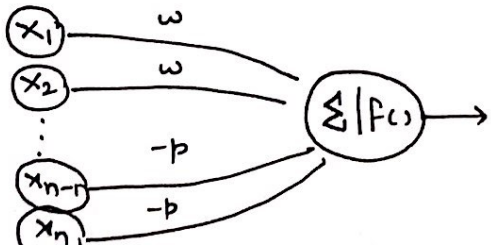
Reinforcement

Bias : Additional input x_0 with value always $=1$ used to enhance or retard the learning
- prevents overfitting / over generalization.

$y = mx + c$ ← Bias
↳ ANN out

McCulloh-Pitts Neuron

- Binary activation and the wts may be +ve or -ve.
- all the +ve wts into a neuron will be the same. (called excitary)
- all the -ve wts into a neuron will be the same (called existential).



in general keep the threshold θ as:

$\theta \geq nw - p$

Hebb Network

- change in the synaptic gap causes the learning.
- When an axon of A is near enough to excite B and repeatedly / permanently takes place in firing it. Some Metabolic change occurs in one or both A & B s.t A's efficiency as one of the cells firing B increases.
- made to work on bipolar data.

$$w_i(new) = w_i(old) + x_i y$$

wt updation Rule.

$$b_{new} = b_{old} + y$$

bias update rule.

$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{b}{w_2}$$

Training Algorithm

1. initialize the wts to zero.
2. do for all inputs
   1. input nodes have the identity activation
   2. output are set to target
   3. adjust wts and bias

Pick the wts which satisfy most of the inputs as final

AND : 2, 2, -2.

OR , 2, 2, +2.

Non Monotonic Reasoning-

- Logic based systems are monotonic in nature i.e if a proposition is made which is true, it remains true for all circumstances.
- Real life systems are not monotonic.
- Monotonic logic only works when:
  1. Information is complete.
  2. Info is consistent
  3. It can only change if new info is consistent with all others.
- Monotonic Systems fail IRL because:
  1. Info is always incomplete.
  2. Situations keep changing ∴ so do the soln's.
  3. assumptions are made for default cases (not IRL).

Issues :

- How can KB be extended to make decision on basis of lack of knowledge as well as presence.
- How can KB be updated. when there is entry/exit.
- How can KB be used when there are several conflicts.

NM Reasoning-

1. NM Logic

- Augment FOL with operator M

$$\forall x,y : R(x,y) \wedge M\, G(x,y) \rightarrow \neg W(x,y)$$

for all x and related y if both get-along and this is consistent then conclude.

issue : what is consistent?
: what to do when statements taken alone arrive at diff result.
↳ Cant Conclude.

- Dependency directed backtracking solves alot of issues of NML.
- Autoepistemic logic solves some issues.

Default Logic

$$\frac{A : B}{C}$$

If A is provable and is consistent with B ; conclude C.

- The rules of inference on basis of comp, using this we can get possible extension of KB.
- Some rules cannot be manip by others.

- **Inheritance**

→ Add Abnormalities to keep inconsistencies in check.

Expert Systems

- They are knowledge intensive computer programs that capture the knowledge and thinking of human experts and then simulate it to solve various problems.
- Also called AI / Knowledge Based Systems.
- To solve the jobs done by human experts, ES need access to substantial domain knowledge base (which is built as efficiently as possible)
- The Expert Sys exploits one or more reasoning mechanism to apply the knowledge to given problem.

Characteristics of ES

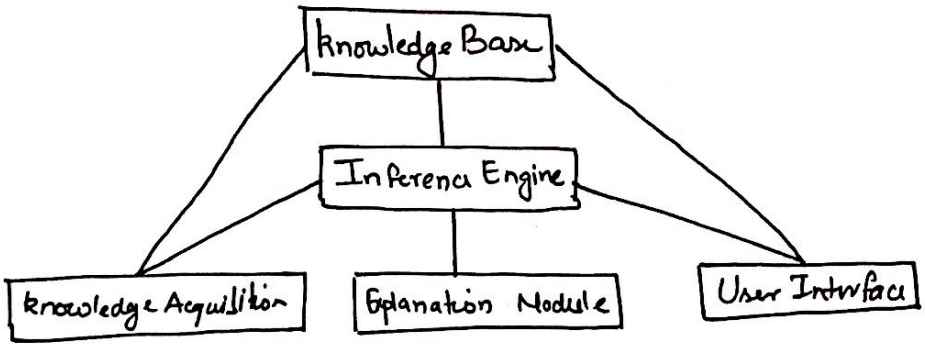
- Solve difficult problems as good as or better than humans.
- They posses vast qualities of domain specific knowledge to the minute detail. (eg to knowledge from work experience)
- Permit use of Heuristic Search procss. i.e Build it in.
- Be able to deal with uncertain and irrelevant data.
- Communicate with the users using natural language.
- They should have a high return-on-investment.

Need of ES

- Human experts are a scarse resource and are not available at all times
- Humans make errors, they are not 100% reliable or consistent.
- Human experts may be able to solve problems but they may not be able to explain the solution.
- Cost of ES << Cost of hiring Human Expert.

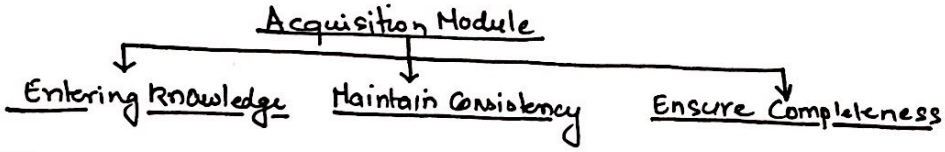
Fundamental Capabilities : People must be able to interact with them.

1. **Explain It's Reasoning**: People may not accept soln if they aren't 100% convinced behind its reasoning. ∴ ES must be able to Explain the Reasoning Mechanism it used to get to that solution.
2. **Acquire New Knowledge & Modify Old**: The power to solve problems comes from the knowledge base ∴ ES must continuously refine them and acquire new intel to update itself.



Components

1. **Knowledge Acquisition**: Process of collection and organization of new knowledge
   - A Knowledge Elicitation: Knowledge elicited by Engineers by interviewing Experts.
   - B Knowledge Representation: Format to represent in an ~~eff~~ manner for efficient and usable utilization. The output is the knowledge base.



Eg MOLE, SALT

2 **Inference Engine**: Contains the general learning algorithm and rules to exploit the knowledge base.

3. **Explanation Module**: Defines the ability to explain itself.
   : Provide UI for good explanation to justify decision.

4. **Knowledge Base**: This is the core module. i.e The warehouse of domain knowledge. The knowledge can be in logics, frames, nets etc.

5. **User Interface**: Provides the Human Interaction tools. like
   - consultation to remedy problems.
   - Knowing put info of system.
   - get explanation.

Applications: Medicine : diagnosis, testing, Diagnosis of electronics, Diag of S/w tools, forecasting crop damage, fault locating, loan evaluations, military planning and exploration.

Examples

1. PUFF : Medicine : Diagnosis of Repp conditions
2. PROSPECTOR : Mining : To locate site for Drilling / Mining.
3. MYCIN : Medicine : Diagnosis of Blood Disorders.
4. DesignAdvisor : Industry : Design of electronic CHips.
5. DENDRAL : Scienca : Structure of Chem compounds
6. LITHIAN : archeology : Examining stone tools.

Problems With ES

1. Brittleness : They only have highly domain specific knowledge & Lack a fallback to normal GK.

   Eg Two data items getting reversed during entry.

2. Lack of Meta Knowledge : They have no clue about themselves. No system Exist to help an ES explain itself.

3. Knowledge Acquisition : This is a major bottle neck in applying ES to new domains.

4. Validations : It is difficult to measure performance because we cant quantify the use knowledge.
   - diff to quantitatively measure how much knowledge is used.
   - one way is to compare it to a hooman.