Artificial Intelligence

It is the study of how to make computers do things which, at the moment, people do better. It leads FOUR important categories.

i) Systems that think like humans
ii) Systems that act like humans
iii) Systems that think rationally
iv) Systems that act rationally

Acting humanly:

The Turing Test approach: To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask questions of either the person or the computer by typing questions and receiving typed responses. However, the interrogator knows them only as A and B and aims to determine which the person is and which are the machine. The goal of the machine is to fool the interrogator into believing that is the person. If the machine succeeds at this, then we will conclude that the machine is acting humanly. But programming a computer to pass the test provides plenty to work on, to posses the following capabilities.

Thinking Humanly:

The Cognitive modeling approach: To construct a machine program to think like a human, first it requires the knowledge about the actual workings of human mind. After completing the study about human mind it is possible to express the theory as a computer program. It the program’s input/output and timing behavior matches with the human behavior then we can say that the program’s mechanism is working like a human mind.

Example: General Problem Solver (GPS)

Thinking rationally:

The laws of thought approach: The right thinking introduced the concept of logic.

Example: Ram is a student of III year CSE. All students are good in III year in CSE. Ram is a good student

ACTING RATIONALLY:

Acting rationally means, to achieve one’s goal given one’s beliefs. In the previous topic laws of thought approach, correct inference is selected, conclusion is derived, but the agent acts on the conclusion defined the task of acting rationally.

AGENT

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

A rational agent is one that does the right thing.
We use the term **performance measure** for the *how*—the criteria that determine how successful an agent is. Obviously, there is not one fixed measure suitable for all agents.

**agent program**: a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of ARCHITECTURE( computing device). Hence a Agent is a combination of Architecture and Program.

\[
\text{agent} = \text{architecture} + \text{program}
\]

**The Skeleton of an agent is**

```plaintext
function SKELETON-AGENT( percept) returns action

static: memory, the agent’s memory of the world
memory <- UPDATE-MEMORY(memory, percept)
action <- CHOOSE-BEST-ACTION(memory)
memory <- UPDATE-MEMORY(memory, action)
return action
```

A Table driven Agent function:

```plaintext
function TABLE-DRIVEN-AGENT( percept) returns action

static: percepts, a sequence, initially empty
table, a table, indexed by percept sequences, initially fully specified
append percept to the end of percepts
action <- LOOKUP( percepts, table)
return action
```

Four types of agent program:

1. Simple reflex agents
2. Agents that keep track of the world
3. Goal-based agents
4. Utility-based agents

**Simple reflex agents:**

Structure of a simple reflex agent in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.
**function** SIMPLE-REFLEX-AGENT( *percept*) **returns** action

**static**: rules, a set of condition-action rules

\[ \text{state} <- \text{INTERPRET-INPUT}(*\text{percept}) \]
\[ \text{rule} <- \text{RULE-MATCH(state, rules)} \]
\[ \text{action} <- \text{RULE-ACTION}[\text{rule}] \]

**return** action
2. Agents that keep track of the world (Reflex Model)

The simple reflex agent described before will work only if the correct decision can be made on the basis of the current percept. In order to choose an action sometimes INTERNAL STATE will help to take good decision.

```
function REFLEX-AGENT-WITH-STATE( percept ) returns action
static: state, a description of the current world state
rules, a set of condition-action rules
state <- UPDATE-STATE(state, percept)
rule <- RULE-MATCH(state, rules)
action <- RULE-ACTION[rule]
state <- UPDATE-STATE(state, action)
return action
```

3. Goal-based agents

Knowing about the current state of the environment is not always enough to decide what to do. In other words, as well as a current state description, GOAL the agent needs some sort of goal information, which describes situations that are desirable.

4. Utility Based Model:

Goals alone are not really enough to generate high-quality behavior. For example, there are many action sequences that will get the taxi to its destination, thereby achieving the goal, but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude distinction between “happy” and “unhappy” states, whereas a more general performance measure should allow a comparison of different world states. Then it has higher utility for the agent.
Search:
Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states.

- A state is defined by the specification of the values of all attributes of interest in the world
- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The initial state is where you start
- The goal state is the partial description of the solution

State Space Search Notations
The set of notations involved in the state space search is:
1) An initial state is the description of the starting configuration of the agent
2) An action or an operator takes the agent from one state to another state which is called a successor state. A state can have a number of successor states.
3) A plan is a sequence of actions. The cost of a plan is referred to as the path cost.

Problem formulation & Problem Definition
Problem formulation means choosing a relevant set of states to consider, and a feasible set of operators for moving from one state to another.
Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.
A search problem consists of the following:
- S: the full set of states
- s0 : the initial state
- A:S→S is a set of operators
- G is the set of final states. Note that G ⊆ S
The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g∈G.
A simple search procedure is
Do until a solution is found or the state space is exhausted.
1. Check the current state
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state
   If it is not, the new state becomes the current state and the process is repeated

Basic Search Algorithm is
Let L be a list containing the initial state (L= the fringe)
Loop if L is empty return failure
Node <- select (L)
if Node is a goal
then return Node (the path from initial state to Node)
else generate all successors of Node, and merge the newly generated states into L
End Loop
Factors to measure the search Algorithms:
1. Complete
2. Optimal
3. What is the search cost associated with the time and memory required to find a solution?
   a. Time complexity
   b. Space complexity
The different search strategies that we will consider include the following:
1. Blind Search strategies or Uninformed search
   a. Depth first search
   b. Breadth first search
   c. Iterative deepening search
   d. Iterative broadening search
2. Informed Search
3. Constraint Satisfaction Search
Blind search:
Blind search or uninformed search that does not use any extra information about the problem domain. The two common methods of blind search are:
• BFS or Breadth First Search
• DFS or Depth First Search
Search Tree is helpful for the searching the goal node.
The terminologies involved in the search tree is
Root Node: The node from which the search starts.
Leaf Node: A node in the search tree having no children.
Ancestor/Descendant: X is an ancestor of Y is either X is Y’s parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
Branching factor: the maximum number of children of a non-leaf node in the search tree
Path: A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

**Breadth First Search**

Let *Queue* be a list containing the initial state
Loop if *Queue* is empty return failure Node  remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else generate all successors of Node, and
(merge the newly generated nodes into *fringe*)
add generated nodes to the back of *Queue*
End Loop

Advantages of Breadth First Search
Finds the path of minimal length to the goal.
Disadvantages of Breadth First Search
Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

**Depth First Search**

Let *Queue* be a list containing the initial state
Loop if *Queue* is empty return failure
Node<-remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else generate all successors of Node,
and merge the newly generated nodes into *Queue*
add generated nodes to the front of *Queue*
End Loop

Depth Limited Search
Let fringe be a list containing the initial state
Loop if fringe is empty return failure
Node<-remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else if depth of Node = limit return cutoff
else add generated nodes to the front of fringe
End Loop

Depth-First Iterative Deepening (DFID)
until solution found do
*DFS with depth cutoff c*
\[ c = c + 1 \]
A* algorithm.
\[ f(n) = g(n) + h(n) \]
where \( g(n) \) = sum of edge costs from start to \( n \)
\( h(n) \) = estimate of lowest cost path from
\( f(n) \) = actual distance
\( h(n) \) is said to be admissible if it underestimates the cost reached from \( n \).
If \( C^*(n) \) is the cost of the cheapest
\( h(n) \leq C^*(n) \) we can prove that if \( h(n) \) is admissible, then the search will find an optimal solution.

**A* Algorithm:**

OPEN = nodes on frontier.
CLOSED = expanded nodes.
OPEN={<S,NIL>} while OPEN is not empty place <n,p>

**REMOVE FROM OPEN NODE**<n,p>**with minimum** \( f(n) \)

if \( n \) is a goal node,
return success (path p)
for each edge connecting \( n \) & \( M \) with cost c
if <\( m, q > \) is on CLOSED and \( \{p|e\} \) is cheaper than \( q \)
then remove \( n \) from CLOSED,
put <\( m,\{p|e\}\) on OPEN
else if <\( m,q \) is on OPEN and \( \{p|e\} \) is cheaper than \( q \)
then replace \( q \) with \( \{p|e\} \)
else if \( m \) is not on OPEN
then put <\( m,\{p|e\}\) on OPEN
return failure
Here the Starting node is S and the goal node is represented as G. Our aim is to reach the goal node G by tracing out the algorithm described.

**Status of OPEN and CLOSED list after each step:**

NOTE: OPEN list and CLOSED list will be abbreviated as OL and CL from henceforth.
Whenever a node enters the OL, the value of its parent node and its total cost from the starting node S will be included within simple braces().

*Step 1:*
OL: S ( NULL ; 0) CL: (NULL)

*Step 2:*
OL: A (S,1) , B(S,3) , C(S,10) CL:S

*Step 3:*
OL: B(S,3), C(S,10), D(A,6) CL: S,A
Step 4:  
OL: C(S,10), D(A,6), E(B,7)  CL: S,A,B  
Step 5:  
OL: D(A,6), E(B,7)  CL: S,A,B,C  
Step 6:  
OL: E(B,7), F(D,8)  CL: S,A,B,C,D  
Step 7:  
OL: F(D,8), G(E,14)  CL: S,A,B,C,D,E  
Step 8:  
OL: G(E,14)  CL: S,A,B,C,D,E,F  
Step 9:  
OL: --  CL: S,A,B,C,D,E,F,  

**Constraint satisfaction problems** or **CSPs** are mathematical problems where one must find states or objects that satisfy a number of **constraints** or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

A **Constraint Satisfaction Problem (CSP)** is characterized by:  
a set of **variables** \( \{x_1, x_2, \ldots, x_n\} \),  
for each variable \( x_i \) a **domain** \( D_i \) with the possible values for that variable, and  
a set of **constraints**, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each \( i \) from 1 to \( n \), a value in \( D_i \) for \( x_i \) so that all constraints are satisfied.

A CSP can easily be stated as a sentence in first order logic, of the form:  
\[
(\exists x_1) \ldots (\exists x_n) (D_1(x_1) \& \ldots D_n(x_n) \Rightarrow C_1 \ldots C_m)
\]
- What is a CSP?  
  - Finite set of variables \( V_1, V_2, \ldots, V_n \)  
  - Nonempty domain of possible values for each variable \( D_{V_1}, D_{V_2}, \ldots, D_{V_n} \)  
  - Finite set of constraints \( C_1, C_2, \ldots, C_m \)  
- Each constraint \( C_i \) limits the values that variables can take,  
  - e.g., \( V_1 \neq V_2 \)  
- A **state** is defined as an **assignment** of values to some or all variables.  
- **Consistent assignment**  
  - assignment does not violate the constraints

- **CSP benefits**

- Standard representation pattern
– Generic goal and successor functions

– Generic heuristics (no domain specific expertise).
  • An assignment is complete when every variable is mentioned.
  • A solution to a CSP is a complete assignment that satisfies all constraints.

Some CSPs require a solution that maximizes an objective function

Variables: WA, NT, Q, NSW, V, SA, T

• Domains: \( D_i = \{ \text{red}, \text{green}, \text{blue} \} \)

• Constraints: adjacent regions must have different colors.
  • E.g. WA NT
  • Solutions are assignments satisfying all constraints, e.g.
    \( \{ \text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green} \} \)

• Constraint graph:
  • nodes are variables
  • arcs are binary constraints
Graph can be used to simplify search

e.g. Tasmania is an independent subproblem

**Types Of Variables:**
Discrete variables
- Finite domains; size $d O(dn)$ complete assignments.
- E.g. Boolean CSPs: Boolean satisfiability (NP-complete).
- Infinite domains (integers, strings, etc.)
- E.g. job scheduling, variables are start/end days for each job

- Need a constraint language e.g $StartJob1 + 5 \leq StartJob3$.

- Infinitely many solutions

- Linear constraints: solvable

- Nonlinear: no general algorithm

Continuous variables
- E.g. building an airline schedule or class schedule.

- Linear constraints solvable in polynomial time by LP methods.

Unary constraints involve a single variable.
- E.g. $SA$ green

Binary constraints involve pairs of variables.
- E.g. $SA$ $WA$
**CSP as SEARCH:**
- A CSP can easily be expressed as a standard search problem.

- Incremental formulation
  - *Initial State:* the empty assignment {}
  - *Successor function:* Assign a value to any unassigned variable provided that it does not violate a constraint
  - *Goal test:* the current assignment is complete
  - *Path cost:* constant cost for every step

CSP By BackTracking:
- Similar to Depth-first search

  - Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

  - Uninformed algorithm **function** BACKTRACKING-SEARCH(csp) **return** a solution or failure

    **return** RECURSIVE-BACKTRACKING({}, csp)

    **function** RECURSIVE-BACKTRACKING(assignment, csp) **return** a solution or failure

    if assignment is complete then **return** assignment

    var SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)

    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do

      if value is consistent with assignment according to CONSTRAINTS[csp] then add {var=value} to assignment

      **result** RRECURSIVE-BACKTRACKING(assignment, csp)

      if result failure then **return** result

      remove {var=value} from assignment

    return failure
Minimum Remaining Values:

\[ \text{var SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)} \]

- A.k.a. most constrained variable heuristic

- **Heuristic Rule**: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values

**Forward Checking:**

- Can we detect inevitable failure early?
  - And avoid it later?

- **Forward checking idea**: keep track of remaining legal values for unassigned variables.

- Terminate search when any variable has no legal values.
- Assign \( \{WA=\text{red}\} \)
- Effects on other variables connected by constraints to WA
  - \( NT \) can no longer be red
  - \( SA \) can no longer be red
- If $V$ is assigned blue
- Effects on other variables connected by constraints with WA
  - NSW can no longer be blue
  - SA is empty
  - FC has detected that partial assignment is inconsistent with the constraints and backtracking can occur.

Arc Consistency:

- An Arc $X \rightarrow Y$ is consistent if
  for every value $x$ of $X$ there is some value $y$ consistent with $x$

  (note that this is a directed property)
- Consider state of search after WA and Q are assigned:
  SA $\rightarrow$ NSW is consistent if