1 Introduction and History

1.1 Intelligent Agents

- **A First Course in Artificial Intelligence** by Deepak Khemani. First 7 chapters will be covered.
- **Intelligent Agents**: An entity that is persistent(it's there all the time), autonomous, proactive(decide what goals to achieve next) and goal directed(once it has goals, it will follow them). Human beings are also agents.
- An intelligent agent in a world carries a model of the world in its "head". the model may be an abstraction. A self-aware agent would model itself in the world model.
- Signal→Symbol→Signal
- Sense(Signal Processing) → Deliberate(Neuro fuzzy reasoning + Symbolic Reasoning) → Act(Output Signal)

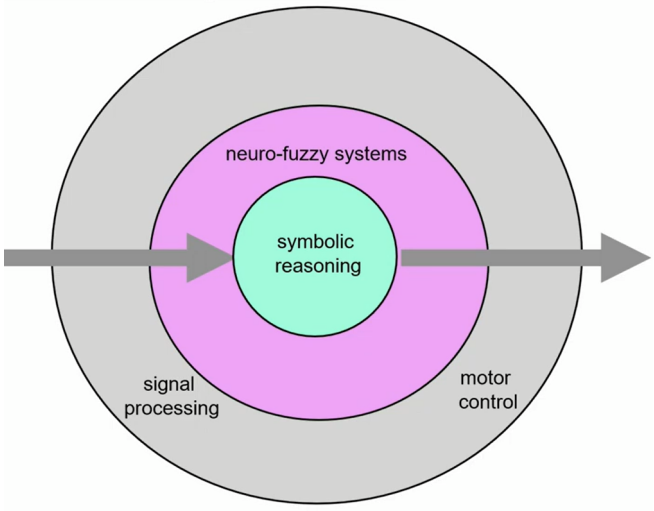


Figure 1: Information Processing View of AI

- **Intelligence**: **Remember the past and learn from it.** Memory and experience(case based reasoning), Learn a model(Machine learning), Recognize objects, faces, patterns(deep neural networks).
  Understand the present. Be aware of the world around you. Create a model of the world(knowledge representation), Make inferences from what you know(logic and reasoning).
  Imagine the future. Work towards your goals. Trial and error(heuristic search), Goals, plans, and actions(automated planning).

1.2 Human Cognitive Architecture

- **Knowledge and Reasoning**: What does the agent know and what else does the agent know as a consequence of what it knows.
- **Semiotics**: A symbol is something that stands for something else. All languages, both spoken and written, are semiotic systems.
- **Biosemiotics**: How complex behaviour emerges when simple systems interact with each other through signs.
- **Reasoning**: The manipulation of symbols in a meaningful manner.

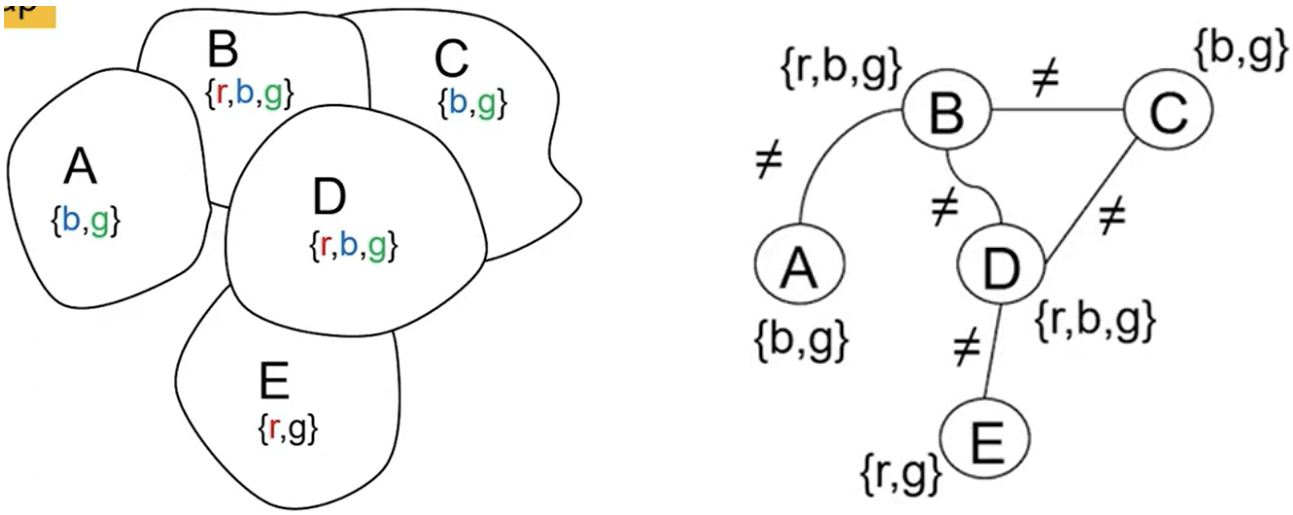
1.3 Problem-Solving

- An autonomous agent in some world has a goal to achieve and a set of actions to choose from to strive for the goal.
- We deal with simple problems first, i.e., the world is static, the world is completely known, only one agent changes the world, action never fail, representation of the world is taken care of.

2 Search Methods

2.1 State Space Search

- We start with the map coloring problem, where we want to color each region in the map with an allowed color such that no two adjacent regions have the same color.



(a) Map with Constraints

(b) Constraint Graph

Figure 2: Map Coloring Problem

Brute Force: Try all combinations of color for all regions.
Informed Search: Choose a color that does not conflict with neighbors.
General Search: Pose the problem to serve as an input to a general search algorithm.
Map coloring can be posed as a constraint satisfaction problem or as a state space search, where the move is to assign a color to a region in a given partially colored state.

- **General Purpose methods**: Instead of writing a custom program for every problem to be solved, these aim to write search algorithms into which individual methods can be plugged in. One of them is State Space Search.
- **State or Solution Space Search**: Describe the given state, devise an operator to choose an action in each state, and then navigate the state space in search of the desired or goal state. Also known as Graph Search.
  A **state** is a representation of a situation.
  A *MoveGen* function captures the moves that can be made in a given state.
  It returns a set of states - *neighbors* - resulting from the moves.
  The state space is an implicit graph defined by the *MoveGen* function.
  A *GoalTest* function checks if the given state is a *goal* state.
  A *search algorithm* navigates the state space using these two functions.
- **The Water Jug Problem**: You have three jugs with capacity 8, 5 and 3 liters. Any state can be written as $[a, b, c]$, where a, b, c are the amount of water present in each jug
  Start state: The 8-liter jug is filled with water, and the other two are empty, $[8, 0, 0]$. It can be seen that at any state the total amount of water remains the same.
  Goal: You are required to measure 4 liters of water. So, our goal state is $[4, x, y]$ or $[x, 4, y]$ or $[x, y, 4]$. A *GoalTest* function can be written as

Algorithm 1 GoalTest for the Water Jug Problem

Require: $[a, b, c]$, the state which needs to be checked
1: **if** $a = 4$ or $b = 4$ or $c = 4$ **then**
2: **return** True
3: **end if**
4: **return** False

- A few other examples are **The Eight Puzzle**, **The Man, Goat, Lion, Cabbage** and **The N-Queens Problem**.

2.2 General Search Algorithms

- Searching is like treasure hunting. Our approach would be to generate and test where we traverse the space by generating new nodes and test each node whether it is the goal or not.
- **Simple Search 1**: Simply pick a node N from OPEN and check if it is the goal.

Algorithm 2 Simple Search 1

1: ▷ S is the initial state
2: OPEN ← $\{S\}$
3: **while** *OPEN* is not empty **do**
4: pick some node N from OPEN
5: OPEN ← OPEN - $\{n\}$
6: **if** GoalTest(N) = True **then**
7: **return** N
8: **else**
9: OPEN ← OPEN ∪ MoveGen(N)
10: **end if**
11: **end while**
12: **return** FAILURE

This algorithm may run into an infinite loop, to address it we have **Simple Search 2**.

Algorithm 3 Simple Search 2

1: ▷ S is the initial state
2: OPEN ← $\{S\}$
3: CLOSED ← $\{\}$
4: **while** *OPEN* is not empty **do**
5: pick some node N from OPEN
6: OPEN ← OPEN - $\{n\}$
7: CLOSED ← CLOSED ∪ $\{N\}$
8: **if** GoalTest(N) = True **then**
9: **return** N
10: **else**
11: OPEN ← OPEN ∪ {MoveGen(N) - CLOSED}
12: **end if**
13: **end while**
14: **return** FAILURE

We can modify this a bit further by doing, OPEN ← OPEN ∪ {*MoveGen(N)* - CLOSED - OPEN}, this lowers the state space even more.

2.3 Planning and Configuration Problems

- **Planning Problems**: Goal is known or describe, path is sought. Examples include River crossing problems, route finding etc.
- **Configuration Problems**: A state satisfying a description is sought. Examples include N-queens, crossword puzzle, Sudoku, etc.
- The simple search algorithms will work for configuration problems, but they won't return any path.

- **NodePairs**: Keep track of parent nodes. Each node is a pair (*currentNode*, *parentNode*).
- **Depth First Search**: OPEN is a stack data structure

Algorithm 4 Depth First Search

```
1: OPEN ← (S, null) : [ ]
2: CLOSED ← [ ]
3: while OPEN is not empty do
4:     nodePair ← head OPEN
5:     (N, _) ← nodePair
6:     if GoalTest(N) = True then
7:         return ReconstructPath(nodePair, CLOSED)
8:     end if
9:     CLOSED ← nodePair : CLOSED
10:    children ← MoveGen(N)
11:    newNodes ← RemoveSeen(children, OPEN, CLOSED)
12:    newPairs ← MakePairs(newNodes, N)
13:    OPEN ← newPairs ++ tail OPEN
14: end while
15: return [ ]

16: function RemoveSeen(nodeList, OPEN, CLOSED)
17:    if nodeList = [ ] then
18:        return [ ]
19:    end if
20:    node ← head nodeList
21:    if OccursIn(node, OPEN) or OccursIn(node, CLOSED) then
22:        return RemoveSeen(tail nodeList, OPEN, CLOSED)
23:    end if
24:    return node : RemoveSeen(tail nodeList, OPEN, CLOSED)
25: end function

26: function OccursIn(node, nodePairs)
27:    if nodePairs = [ ] then
28:        return FALSE
29:    else if node = first head nodePairs then
30:        return TRUE
31:    end if
32:    return OccursIn(node, tail nodePairs)
33: end function

34: function MakePairs(nodeList, parent)
35:    if nodeList = [ ] then
36:        return [ ]
37:    end if
38:    return (head nodeList, parent) : MakePairs(tail nodeList, parent)
39: end function

40: function ReconstructPath(nodePair, CLOSED)
41:    (node, parent) ← nodePair
42:    path ← node : [ ]
43:    while parent is not null do
44:        path ← parent : path
45:        CLOSED ← SkipTo(parent, CLOSED)
46:        (_, parent) ← head CLOSED
47:    end while
48:    return path
49: end function
```

Algorithm 5 Depth First Search continued

```
50: function SkipTo(parent, nodePairs)
51:     if parent = first head nodePairs then
52:         return nodePairs
53:     end if
54:     return SkipTo(parent, tail nodePairs)
55: end function
```

- **Breadth First Search**: We use a queue for the OPEN set.

Algorithm 6 Breadth First Search

```
OPEN ← (S, null) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _) ← nodePair
    if GoalTest(N) = True then
        return ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    children ← MoveGen(N)
    newNodes ← RemoveSeen(children, OPEN, CLOSED)
    newPairs ← MakePairs(newNodes, N)
    ▷ This is the only line that is different, we now add newPairs at the end of the list
    OPEN ← tail OPEN ++ newPairs
end while
return [ ]
```

- BFS always generates the shortest path, whereas DFS may not generate the shortest path.
- **Time Complexity**: Assume constant branching factor b and assume the goal occurs somewhere at depth d. In the best case the goal would be the first node scanned at depth d and in the worst case the goal would b3 the last node scanned.
  Best Case: $N_{DFS} = d + 1$ and $N_{BFS} = \frac{b^d - 1}{b-1} + 1$
  Worst Case: $N_{DFS} = N_{BFS} = \frac{b^{d+1} - 1}{b-1}$
  Average: $N_{DFS} \approx \frac{b^d}{2}$ and $N_{BFS} \approx \frac{b^d(b+1)}{2(b-1)}$, $N_{BFS} \approx N_{DFS}(\frac{b+1}{b-1})$

	Depth First Search	Breadth First Search
Time	Exponential	Exponential
Space	Linear	Exponential
Quality of Solution	No guarantees	Shortest path
Completeness	Not for infinite search space	Guaranteed to terminate if solution path exists

Table 1: DFS vs BFS

- **Depth Bounded DFS**: Do DFS with a depth bound d. It is not complete and does not guarantee the shortest path. Given below is a modified version that returns the count of nodes visited, to get the original version just remove count.

Algorithm 7 Depth Bounded DFS

```
count ← 0
OPEN ← (S, null, 0) : [ ]
CLOSED ← [ ]
while OPEN is not empty do
    nodePair ← head OPEN
    (N, _, depth) ← nodePair
    if GoalTest(N) = True then
        return count, ReconstructPath(nodePair, CLOSED)
    end if
    CLOSED ← nodePair : CLOSED
    if depth < depthBound then
        children ← MoveGen(N)
        newNodes ← RemoveSeen(children, OPEN, CLOSED)
        newPairs ← MakePairs(newNodes, N, depth + 1)
        OPEN ← newPairs ++ tail OPEN
        count ← count + length newPairs
    else
        OPEN ← tail OPEN
    end if
end while
return count, [ ]
```

- **Depth First Iterative Deepening**: Iteratively increase depth bound. Combines best of BFS and DFS. This will give the shortest path, but we have to be careful that we take correct nodes from CLOSED. $N_{DFID} \approx N_{BFD} \frac{b}{b-1}$ for large b.

Algorithm 8 Depth First Iterative Deepening

```
count ← −1
path ← [ ]
depthBound ← 0
repeat
    previousCount ← count
    (count, path) ← DB-DFS(S, depthBound)
    depthBound ← depthBound +1
until (path is not empty) or (previousCount == count)
return path
```

- The monster that AI fights is Combinatorial Explosion.
- All these algorithms we studied are blind algorithms or uniformed search, they don't know where the goal is.

2.4 Heuristic Search

- Testing the neighborhood and following the steepest gradient identifies which neighbors are the lowest or closest to the bottom.
- The heuristic function $h(N)$ is typically a user defined function, $h(Goal) = 0$.
- **Best First Search**: Instead of having a simple stack or queue we use a priority queue sorted based on heuristic function. Search frontier depends upon the heuristic function. If graph is finite then this is complete and quality of solution will head towards the goal but may not give the shortest path.

Algorithm 9 Best First Search

```
1: OPEN ← (S, null, h(S)) : [ ]
2: CLOSED ← [ ]
3: while OPEN is not empty do
4:     nodePair ← head OPEN
5:     (N, _, _) ← nodePair
6:     if GoalTest(N) = True then
7:         return ReconstructPath(nodePair, CLOSED)
8:     end if
9:     CLOSED ← nodePair : CLOSED
10:    children ← MoveGen(N)
11:    newNodes ← RemoveSeen(children, OPEN, CLOSED)
12:    newPairs ← MakePairs(newNodes, N)
13:    OPEN ← sort_h(newPairs ++ tail OPEN)
14: end while
15: return [ ]
```

- For The eight puzzle we can define a few heuristic functions as follows:
  $h_1(n)$ = number of tiles out of place, Hamming distance.
  $h_2(n) = \sum_{\text{for each tile}}$ Manhattan distance to its destination.
- **Hill Climbing**: A local search algorithm, i.e., move to the best neighbor if it is better, else terminate. In practice sorting is not needed, only the best node. This has burnt its bridges by not storing OPEN. We are interested in this because it is a constant space algorithm, which is a vast improvement on the exponential space for BFS and DFS. It's time complexity is linear. It treats the problem as an optimization problem.

Algorithm 10 Hill Climbing

```
1: node ← Start
2: newNode ← head(sort_h(moveGen(node)))
3: while h(newNode) < h(node) do
4:     node ← newNode
5:     newNode ← head(sort_h(moveGen(node)))
6: end while
7: return node
```

2.5 Escaping Local Optima

- **Solution Space Search**: Formulation of the search problem such that when we find the goal node we have the solution, and we are done.
- **Synthesis Methods**: Constructive methods. Starting with the initial state and build the solution state piece by piece.
- **Perturbation Methods**: Permutation of all possible candidate solutions.
- **SAT problem**: Given a boolean formula made up of a set of propositional variables $V = \{a, b, c, d, e, ...\}$ each of which can be *true* or $false$, or 1 or 0, to find an assignment of variables such that the given formula evaluates to *true* or 1.
  A SAT problem with N variables has 2^N candidates.
- **Travelling Salesman Problem**: Given a set of cities and given a distance measure between every pair of cities, the task is to find a Hamiltonian cycle, visiting each city exactly once, having the least cost.
  1. **Nearest Neighbor Heuristic**: Start at some city, move to nearest neighbor as long as it does not close the loop prematurely.
  2. **Greedy Heuristic**: Sort the edges, then add the shortest available edge to the tour as long as it does not close the loop prematurely.
  3. **Savings Heuristic**: Start with $n - 1$ tours of length 2 anchored on a base vertex and performs $n - 2$ merge operations to construct the tour.
  4. **Perturbation operators**: Start with some tour, then choose two cities and interchange, this gives the solution space.

5. **Edge Exchange**: Similar to above but now instead of choosing cities we are choosing edges. Can be 2-edge exchange, 3-edge, 4-edge, etc. City exchange is a special case of 4-edge exchange.

- Solution space of TSP grows in order factorial, much faster than SAT problem.
- A collection of problems with some solutions is available here.
- To escape local minima we need something called **exploration**.
- **Beam Search**: Look at more than one option at each level. For a beam width b, look at best b options. Heavily memory dependent.
- **Variable Neighborhood Descent**: Essentially we are doing hill climbing with different neighborhood functions. The idea is to use sparse functions before using denser functions, so the storage is not high. The algorithm assumes that there are N *moveGen* functions sorted according to the density of the neighborhoods produced.

Algorithm 11 Variable Neighborhood Descent

```
VariableNeighbourhoodDescent( )
node ← start
for i ← 1 to n do
    moveGen ← MoveGen(i)
    node ← HillClimbing(node, moveGen)
end for
return node
```

- **Best Neighbor**: Another variation of Hill Climbing, simply move to the best neighbor regardless of whether it is better than current node or not. This will require an external criterion to terminate. It will not escape local maxima, as once it escapes it will go right back to it in the next iteration.

Algorithm 12 Best Neighbor

```
BestNeighbor( )
N ← start
bestSeen ← N
while some termination criterion do
    N ← best(moveGen(N))
    if N better than bestSeen then
        bestSeen ← N
    end if
end while
return bestSeen
```

- **Tabu Search**: Similar to Best Neighbor but not allowed back immediately. An aspiration criterion can be added that says that if a tabu move result in a node that is better than bestSeen then it is allowed. To drive this search into newer areas, keep a frequency array and give preference to lower frequency bits or bits that have been changed less.

Algorithm 13 Tabu Search

```
TabuSearch( )
N ← start
bestSeen ← N
while some termination criterion do
    N ← best(allowed(moveGen(N)))
    if N better than bestSeen then
        bestSeen ← N
    end if
end while
return bestSeen
```