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)

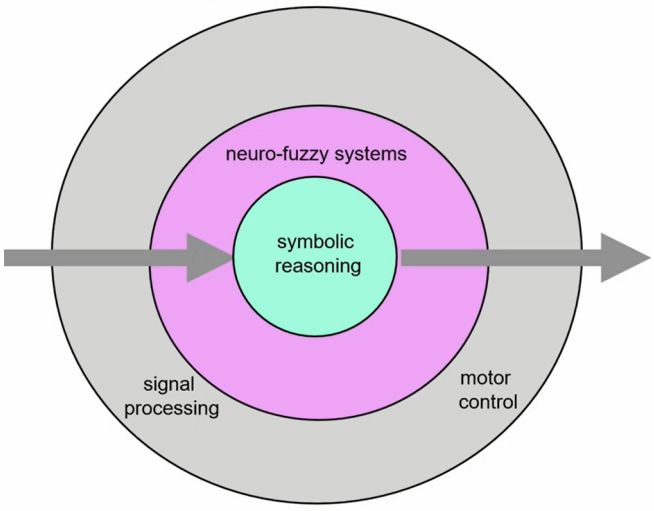


Figure 1: Information Processing View of AI

- **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.

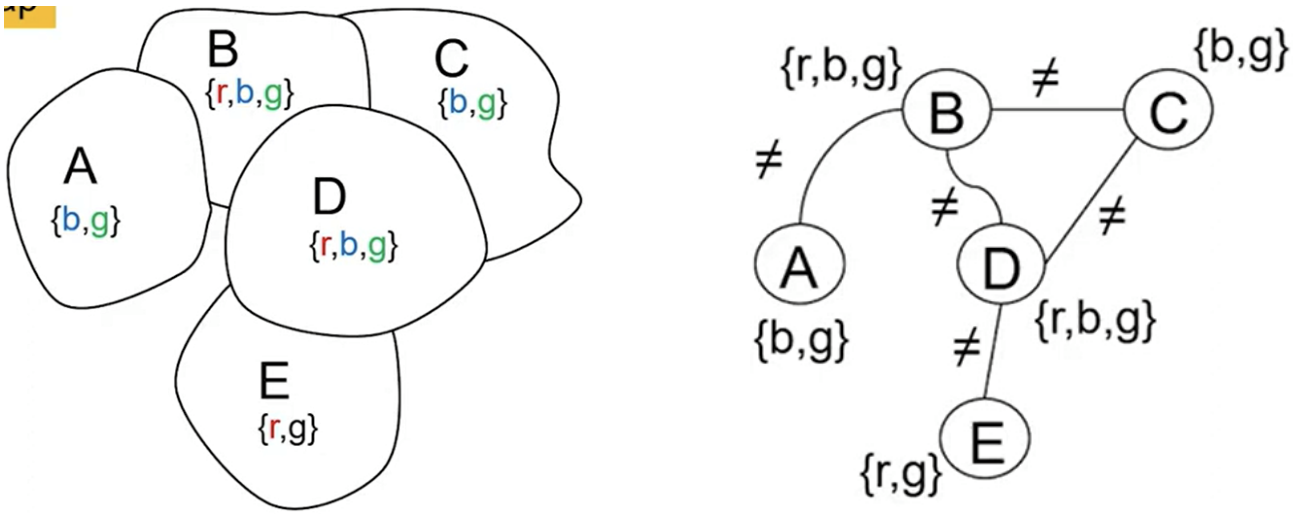
1.3 Problem-Solving

- 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**.

Before studying different algorithms, we need to familiarize ourselves with some pseudocode syntax.

AI SMPS 2022: Lists and Tuples

Version 0.3 (prepared by S. Baskaran)

This is a quick reference for List and Tuple operations used in algorithms discussed in this course. In the assignments and final exam, answers to short-answer-type questions depend on the sequence in which values are added, read and removed from lists and tuples. Therefore, it is important to understand the representation and operations on lists and tuples.

OPERATORS AND EXPRESSIONS

▷	▷ a right pointing triangle starts a line comment
=	▷ equality-test operator
←	▷ assignment operator
:	▷ list constructor, a.k.a, **cons** operator
++	▷ list concatenation operator
null	▷ null value
head	▷ returns the head of a list
tail	▷ returns the tail of a list
take n	▷ returns at most n elements from a list
first	▷ returns the first element of a tuple
second	▷ returns the second element of a tuple
third	▷ returns the third element of a tuple

$\text{expression}_1 = \text{expression}_2$	▷ equality-test expression
pattern ← expression	▷ assignment expression

In what follows, all equality tests ($\text{expr}_1 = \text{expr}_2$) evaluate to true.

LIST OPERATIONS

$\text{LIST}_2 \leftarrow \text{ELEMENT} : \text{LIST}_1$ ▷ list representation

$\text{LIST}_2 \leftarrow \text{HEAD} : \text{TAIL}$ ▷ components of a list

$[]$ ▷ an empty list

$3 : 2 : 1 : []$ ▷ a three element list

$[3, 2, 1]$ ▷ shorthand notation

$[3, 2, 1] = 3 : [2, 1] = 3 : 2 : [1] = 3 : 2 : 1 : []$

$[]$ **is empty** $=$ TRUE

$[1]$ **is empty** $=$ FALSE

$[1] = 1 : []$

$1 =$ **head** $[1] =$ **head** $1 : []$

$[] =$ **tail** $[1] =$ **tail** $1 : []$

(**tail** $[1]$) **is empty** $=$ TRUE

$3 =$ **head** $[3, 2, 1] =$ **head** $3 : 2 : 1 : []$

$[2, 1] =$ **tail** $[3, 2, 1] =$ **tail** $3 : 2 : 1 : []$

$2 =$ **head tail** $[3, 2, 1] =$ **head tail** $3 : 2 : 1 : []$

$[1] =$ **tail tail** $[3, 2, 1] =$ **tail tail** $3 : 2 : 1 : []$

$1 =$ **head tail tail** $[3, 2, 1] =$ **head tail tail** $3 : 2 : 1 : []$

$[o, u, t] =$ **take** 3 $[o, u, t, r, u, n]$

$[a, t] =$ **take** 3 $[a, t]$

$[a] =$ **take** 3 $[a]$

$[] =$ **take** 3 $[]$

$\text{LIST}_3 = \text{LIST}_1 \mathbin{+\!\!+} \text{LIST}_2$

$\text{LIST}_3 \;=\; \text{LIST}_1 \mathbin{+\!\!+} \text{LIST}_2$

$[\,] \;=\; [\,] \mathbin{+\!\!+} [\,]$

$\text{LIST} \;=\; \text{LIST} \mathbin{+\!\!+} [\,] \;=\; [\,] \mathbin{+\!\!+} \text{LIST}$

$[o, u, t, r, u, n] \;=\; [o, u, t] \mathbin{+\!\!+} [r, u, n]$

$[r, u, n, o, u, t] \;=\; [r, u, n] \mathbin{+\!\!+} [o, u, t]$

$[r, o, u, t] \;=\; (\textbf{head}\ [r, u, n]) : [o, u, t]$

$[n, u, t] \;=\; \textbf{tail tail}\ [r, u, n] \mathbin{+\!\!+} \textbf{tail}\ [o, u, t]$

$[n, u, t] \;=\; (\textbf{tail tail}\ [r, u, n]) \mathbin{+\!\!+} (\textbf{tail}\ [o, u, t])$

$a \leftarrow \textbf{head}\ [3, 2, 1]$ ▷ $a \leftarrow 3;$

$b \leftarrow \textbf{tail}\ [3, 2, 1]$ ▷ $b \leftarrow [2, 1];$

$a : b \leftarrow [3, 2, 1]$ ▷ $a \leftarrow 3;\quad b \leftarrow [2, 1];$

$a : b \leftarrow 3 : 2 : 1 : [\,]$ ▷ $a \leftarrow 3;\quad b \leftarrow 2 : 1 : [\,];$

$a : b : c \leftarrow [3, 2, 1]$ ▷ $a \leftarrow 3;\quad b \leftarrow 2;\quad c \leftarrow [1];$

$a : b : c \leftarrow 3 : 2 : 1 : [\,]$ ▷ $a \leftarrow 3;\quad b \leftarrow 2;\quad c \leftarrow 1 : [\,];$

$a : \_ : c \leftarrow [3, 2, 1]$ ▷ $a \leftarrow 3;\quad c \leftarrow [1];$

$a : \_ : c \leftarrow 3 : 2 : 1 : [\,]$ ▷ $a \leftarrow 3;\quad c \leftarrow 1 : [\,];$

TUPLE OPERATIONS

$(101, \text{“Oumuamua”}, 400\text{m})$ ▷ a 3-tuple

$(101, 102)$ ▷ a 2-tuple

$101 \;=\; \textbf{first}\ (101, 102)$

$102 \;=\; \textbf{second}\ (101, 102)$

$\text{pair} \leftarrow (101, 102)$

$101 \;=\; \textbf{first}\ \text{pair} \;=\; \textbf{first}\ (101, 102)$

$102 \quad = \quad \textbf{second}\ \text{pair} \quad = \quad \textbf{second}\ (101, 102)$

$a \quad \leftarrow \quad \textbf{first}\ \text{pair} \qquad\qquad \triangleright\ a \leftarrow 101;$

$b \quad \leftarrow \quad \textbf{second}\ \text{pair} \qquad\qquad \triangleright\ b \leftarrow 102;$

$(a, b) \quad \leftarrow \quad \text{pair} \qquad\qquad \triangleright\ a \leftarrow 101; \quad b \leftarrow 102;$

$(a, b) \quad \leftarrow \quad (101, 102) \qquad\qquad \triangleright\ a \leftarrow 101; \quad b \leftarrow 102;$

$a \quad \leftarrow \quad \textbf{first}\ \text{pair} \qquad\qquad \triangleright\ a \leftarrow 101;$

$(a, \_) \quad \leftarrow \quad \text{pair} \qquad\qquad \triangleright\ a \leftarrow 101;$

$b \quad \leftarrow \quad \textbf{second}\ \text{pair} \qquad\qquad \triangleright\ b \leftarrow 102;$

$(\_, b) \quad \leftarrow \quad \text{pair} \qquad\qquad \triangleright\ b \leftarrow 102;$

$400\text{m} \quad = \quad \textbf{third}\ (101, \text{“Oumuamua”}, 400\text{m})$

$c \quad \leftarrow \quad \textbf{third}\ (101, \text{“Oumuamua”}, 400\text{m}) \qquad\qquad \triangleright\ c \leftarrow 400\text{m};$

$(\_, \_, c) \quad \leftarrow \quad (101, \text{“Oumuamua”}, 400\text{m}) \qquad\qquad \triangleright\ c \leftarrow 400\text{m};$

$101 \quad = \quad \textbf{head second}\ (1, [101, 102, 103], \textbf{null})$

$[102, 103] \quad = \quad \textbf{tail second}\ (1, [101, 102, 103], \textbf{null})$

$(a, h : t, c) \quad \leftarrow \quad (1, [101, 102, 103], \textbf{null})$

$\triangleright\ a \leftarrow 1; \quad h \leftarrow 101; \quad t \leftarrow [102, 103]; \quad c \leftarrow \textbf{null};$

Done. You are ready, now finish your work.

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
```

- DFID-C as compared to regular DFID-N adds new neighbors to OPEN list and also adds neighbors that are in CLOSED but not in OPEN list. This allows DFID-C to get the shortest path whereas DFID-N may not get the shortest 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

```
OPEN ← (S, null, h(S)) : [ ]
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)
    ▷ Again, this is pretty much the most important line
    OPEN ← sort_h(newPairs ++ tail OPEN)
end while
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

```
node ← Start
newNode ← head(sort_h(moveGen(node)))
while h(newNode) < h(node) do
    node ← newNode
    newNode ← head(sort_h(moveGen(node)))
end while
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. Heuristic function is $saving = cost(base, a) + cost(base, b)$ - $cost(a, b)$

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.

Algorithm 11 Beam Search

```
BeamSearch(S, w)
OPEN ← [S]
N ← S
do
    bestEver ← N
    if Goal-Test(OPEN) = True then
        return goal from OPEN
    end if
    neighbors ← MoveGen(OPEN)
    OPEN ← take w sort_h(neighbors)
    N ← head(OPEN)
while h(N) is better than h(bestEver)
return bestEver
```

- **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 12 Variable Neighborhood Descent

```
    VariableNeighbourhoodDescent( )
1:  node ← start
2:  for i ← 1 to n do
3:      moveGen ← MoveGen(i)
4:      node ← HillClimbing(node, moveGen)
5:  end for
6:  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 13 Best Neighbor

```
    BestNeighbor( )
1:  N ← start
2:  bestSeen ← N
3:  while some termination criterion do
4:      N ← best(moveGen(N))
5:      if N better than bestSeen then
6:          bestSeen ← N
7:      end if
8:  end while
9:  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 14 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
```

3 Stochastic Search

Very often to escape local minima we use Stochastic/Randomized methods instead of Deterministic methods.

3.1 Iterated Hill Climbing

- **2SAT**: Problems where each clause has at most two literals, known to be solved in polynomial time. However, if we add even 1 more literal it would become NP-complete or exponential.
- Iterated hill climbing says that we should perform hill climbing with multiple different starting nodes chosen randomly to have a higher probability of finding the goal node.

Algorithm 15 Iterated Hill Climbing

```
Iterated-Hill-Climbing(N)
bestNode ← random candidate solution
for i ← 1 to N do
    currentBest ← Hill-Climbing(new random candidate solution)
    if h(currentBest) is better than h(bestNode) then
        bestNode ← currentBest
    end if
end for
return bestNode
```

3.2 Stochastic Actions

- To move or not to move is the question.
- Idea is we would like to consider actions that are good in terms of heuristic functions, but sometimes we even want to move when heuristic function is not necessarily good.
- **Random Walk**: Choose a random node and move there, hill climbing of stochastic search.

Algorithm 16 Random Walk

```
RandomWalk( )
node ← random candidate solution or start
bestNode ← node
for i ← 1 to n do
    node ← RandomChoose(MoveGen(node))
    if node is better than bestNode then
        bestNode = node
    end if
end for
```

- **Stochastic Hill Climbing**: Let the algorithm be at the current bode v_c, then we select a random neighbor v_N and calculate $\Delta E = (eval(v_N) - eval(v_c))$.
  If ΔE is positive, it means that v_N is better than v_c, then the algorithm should move to it with a higher probability.
  If ΔE is negative, it means that v_N is better than v_c, then the algorithm should move to it with a lower

probability.
The probability is computed using the sigmoid function, which is given as

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

where T is a parameter. The algorithm essentially combines exploration with exploitation.

- **Annealing**: In metallurgy and materials science, **annealing** is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above its recrystallization temperature, maintaining a suitable temperature for an appropriate amount of time and then cooling.
  In annealing, atoms migrate in the crystal lattice and the number of dislocations decreases, leading to a change in ductility and hardness. As the material cools it recrystallizes.
- **Simulated Annealing**: Essentially tries to mimic the annealing process, begins with exploration and ends with exploitation. Annealing is also where ΔE and T come from, essentially meaning Energy and Temperature.

Algorithm 17 Simulated Annealing

```
    SimulatedAnnealing( )
 1: node ← random candidate solution or start
 2: bestNode ← node
 3: T ← some large value
 4: for time ← 1 to numberOfEpochs do
 5:     while some termination criteria do                ▷ M cycles in a sample case
 6:         neighbor ← RandomNeighbor(node)
 7:         ΔE ← Eval(neighbor) − Eval(node)
 8:         if Random(0, 1) < Sigmoid(ΔE, T) then
 9:             node ← neighbor
10:             if Eval(node) > Eval(bestNode) then
11:                 bestNode ← node
12:             end if
13:         end if
14:     end while
15:     T ← CoolingFunction(T, times)
16: end for
17: return bestNode
```

3.3 Genetic Algorithms

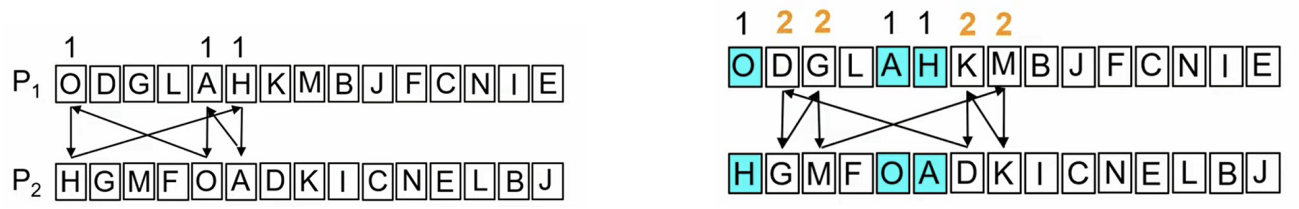
- Survival of the fittest. Inspired by the process of natural selection.
- A class of methods for optimization problems, more generally known as Evolutionary Algorithms.
- Implemented on a population of candidates in the solution space. A fitness function evaluates each candidate. The fittest candidates get to mate and reproduce.
- **Artificial Selection**: Given a population of candidate solutions we do a three step iterative process
  1. **Reproduction**: Clone each candidate in proportion to its fitness. Some may get more than one copy, some none.
  2. **Crossover**: Randomly mate the resulting population and mix up the genes.
  3. **Mutation**: Once in a while, in search of a missing gene.
- **Selection**: Imagine a roulette wheel on which each candidate in the parent population $\{P_1, P_2, ..., P_N\}$ is represented as a sector. The angle subtended by each candidate is proportional to its fitness. The roulette wheel is spun N times. Then the selected parents are added one by one to create a new population.
- **Crossover Operators**: A *single point crossover* simply cuts the two parents at a randomly chosen point and recombines them to form two new solution strings. It can be at multiple points as well, idea is to mix up the genes.

Algorithm 18 Genetic Algorithm

GENETIC-ALGORITHM( )

$P \leftarrow$ create N candidate solutions ▷ initial population
repeat
    compute fitness value for each member of P
    $S \leftarrow$ with probability proportional to fitness value, randomly select N member from P.
    $offspring \leftarrow$ partition S into two halves, and randomly mate and crossover members to generate N offsprings.
    With a low probability mutate some offsprings
    Replace k the weakest members of P with k strongest offsprings
until some termination criteria
return the best member of P

- A large diverse population size is necessary for the performance of genetic algorithm to work.
- One issue is how to represent candidate solutions?, as they can be of any type. One simple solution is to convert the solution into a string this will require an additional function.
- In TSP, one problem is also that during crossover some cities can get repeated.
- **Cycle Crossover**: Identify cycles, look at below image, easier to understand.

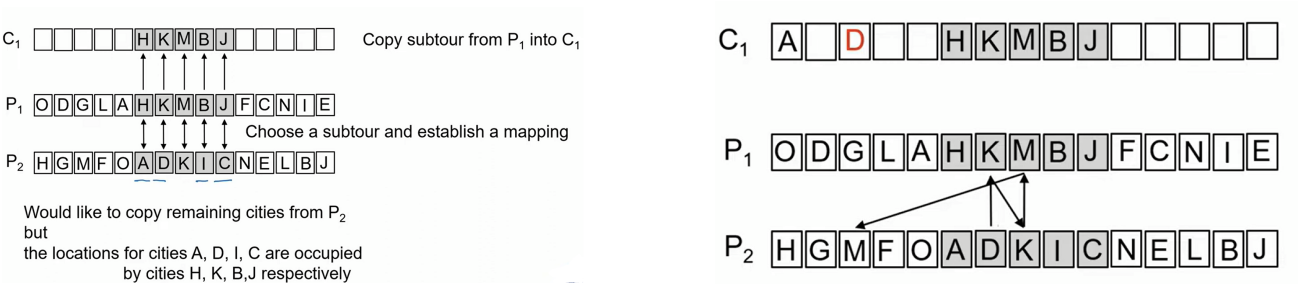


(a) TSP cycle 1 (b) TSP cycle 2

Figure 3: TSP Identifying cycles

Once cycles are identified, then C_1 gets odd numbered cycles from P_1 and even numbered cycles from P_2 the other go to C_2.

- **Partially Mapped Crossover(PMX)**: Identify some cities that form a subtour and establish a mapping.



(a) Choose Subtour (b) Copy occupied spaces

Figure 4: Partially Mapped Crossover

- **Order Crossover**: Copy a subtour from P_1 into C_1 and the remaining from P_2 in the order they occur in P_2.
- For all the above we were using path representation of TSP, there is another representation of TSP.
- **Adjacency Representation**: The cities are arranged based on where they come from in the tour with respect to the index. For example if $A \rightarrow H$, then at index 0 we would have H.
- **Alternating Edges Crossover**: From a given city X choose the next city Y from P_1 and from the city Y choose the next city from P_2 and so on. It is possible to run into cycles, need to be careful.
- **Heuristic Crossover**: For each city choose the next from that parent P_1 or P_2 whichever is closer.

- **Ordinal Representation**: Replace the name of the city is Path Representation one by one to its current numeric index. At the start all cities A, B, C, D, E will have numeric index $1, 2, 3, 4, 5$, now consider if we add C to the representation the new indexing would be for cities A, B, D, E, we have $1, 2, 3, 4$. The **advantage** is that single point crossover produces valid offspring.

3.4 Ant Colony Optimization

- **Emergent Systems**: Collections of simple entities organize themselves and a larger more sophisticated entity emerges. The behavior of this complex system is a property that emerges from interactions amongst its components.
- **Conway's Game of Life**: A cellular automaton in which cells are alive or dead. Each cell obeys the following rules to decide its fate in the next time step.

Cell state	Number of alive neighbors	New cell state
alive	< 2	dead
alive	2 or 3	alive
alive	> 3	dead
dead	3	alive

Table 2: Rules for Game of Life

- Illusion of movement can be seen in an example called **Gosper's Glider Gun**.
- **Chaos and Fractals**: A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop. Drive by recursion, fractals are images of dynamic systems, the pictures of chaos.
- Let 5 ants A, B, C, D, and E go out in search for food. Each ant lays a trail of pheromone where it goes. Each ant lays a trail of pheromone where it goes. More ants that emerge will tend to follow some pheromone trail.
- Let's say A finds some food, then it will follow its pheromone trail back and the other ants will continue their search.
- Eventually, as more ants travel on the trail they deposit more pheromone and the trail gets stronger and stronger, eventually becoming the caravan we might have seen raiding our food.
- They tend to find the shortest path.
- **Ant Colony Optimization**: We try to capitalize on this method. Each ant constructs a solution using a stochastic greedy method using a combination of a heuristic function and pheromone trail following.
- This is related to the class of algorithms known as *Swarm Optimization*.

Algorithm 19 Ant Colony Optimization for TSP

```
    TSP-ACO( )
 1: bestTour ← NIL
 2: repeat
 3:     randomly place M ants on N cities
 4:     for each ant a do
 5:         for n ← 1 to N do
 6:             ant a selects an edge from the distribution P_n^a
 7:         end for
 8:     end for
 9:     update bestTour
10:     for each ant a do
11:         for each edge (u, v) in the ant's tour do
12:             deposit pheromone ∝ 1/tour-length on edge (u, v)
13:         end for
14:     end for
15: until some termination criteria
16: return bestTour
```

- From a city i the k^{th} ant moves to city j with a probability given by

$$P_{ij}^{k}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} \times [\eta_{ij}]^{\beta}}{\sum_{h \in allowed_k(t)} ([\tau_{ih}(t)]^{\alpha} [\eta_{ih}(t)]^{\beta})}, & \text{if } j \in allowed_k(t) \text{ the cities ant } k \text{ is allowed to move to} \\ 0, & \text{otherwise} \end{cases}$$

  where $\tau_{ij}(t)$ is pheromone on edge$_{ij}$ and η_{ij} is called visibility which is inversely proportional to the distance between cities i and j.

- After constructing a tour in n time steps, each ant k deposits an amount of pheromone $\frac{Q}{L_k}$ on the edges it has traversed, which is inversely proportional to the cost of the tour L_k it found.

- Total pheromone deposited on edge$_{ij}$ is $\Delta\tau_{ij}(t, t+n)$

- The total pheromone on edge$_{ij}$ is updated as

$$\tau_{ij}(t+n) = (1-\rho) \times \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n)$$

  where ρ is the rate of evaporation of pheromone.

4 Finding Optimal TSP Tours

4.1 Finding Optimal Paths

- Breadth First Search finds a solution with the smallest *number* of moves. But if *cost* of all moves is no the same then an **optimal** solution may not be the one with the smallest number of moves.
- **Brute Force**: Simply search the entire search tree, computationally it is mindlessly expensive.
- **Goal**: Search as little of the space as possible while guaranteeing the optimal solution.
- A *Best First* approach to solving problems. Given a set of candidates a search algorithm has to choose from. Each candidate is tagged with an estimated cost of the complete solution.
- Branch and Bound in a refinement space
  1. Initial solution set contains all solutions.
  2. In each step we partition a set into two smaller sets.
  3. Until we can pick a solution that is fully specified.
  4. Each solution set needs to have an estimated cost.
  5. B&B will refine the solution that has the least estimated cost.
- An optimal solution can be guaranteed by ensuring that the estimated cost is a lower bound on actual cost.
- **Lower bound**: A solution will never be cheaper than it is estimated to be.
- Thus if a fully refined solution is cheaper than a partially refined one, the latter need not be explored - **PRUNING**. The higher the estimate, the better the pruning.
- Branch and Bound for TSP
  1. Let the candidate solutions be permutations of the list of city names.
  2. The initial set of candidates includes all permutations.
  3. Refine the cheapest set by specifying a specific segment.
  4. Heuristic: Choose the segment with minimum cost.
  5. For tighter estimates, edges that cause three or more segment cycles are avoided.
  6. We, can also exclude an edge that is a third edge for a node, because a city has only two neighbors in a tour.
  7. Higher estimates require more work.
- A lower bound estimate could be for each row add up the smallest two positive entries and divide by two. This may not feasible.
- The basic idea behind B&B is to prune those parts of a search space which cannot contain a better solution.
- Each candidate is tagged with an estimated cost of the complete solution.

- **Dijkstra's Algorithm**

  1. Begins by assigning infinite cost estimates to all nodes except the start node.
  2. It assigns color white to all the nodes initially.
  3. It picks the cheapest white node and colors it black.
  4. **Relaxation**: Inspect all neighbors of the new black node and check if a cheaper path has been found to them.
  5. If yes, then update cost of that node, and mark the parent node.

4.2 Algorithm A*

- A* achieves better performance by using heuristics to guide its search.
- For all nodes we will compute $f(n)$ which is made up of two components $g(n)$ and $h(n)$.
- $g(n)$ = actual cost of solution found from the start node to node n.
- $h(n)$ = estimated cost of the path from node n to goal node.
- Maintain a priority queue of nodes sorted on the f values
- Pick the lowest f value node, and check whether it is the goal node.
- Else generate its neighbors, and compute their f values
- Insert new nodes into the priority queue
- For existing nodes check for better path(like Dijkstra's algorithm)

Algorithm 20 A* Algorithm

```
   A*(S)
 1: default value of g for every node is +∞
 2: parent(S) ← null
 3: g(S) ← 0
 4: f(S) ← g(S) + h(S)
 5: OPEN ← S :[ ]
 6: CLOSED ← empty list
 7: while OPEN is not empty do
 8:     N ← remove node with the lowest f value from OPEN
 9:     add N to CLOSED
10:     if GoalTest(N) then
11:         return ReconstructPath(N)
12:     end if
13:     for each neighbor M ∈ MoveGen(N) do
14:         if g(N) + k(N, M) < g(M) then
15:             parent(M) ← N
16:             g(M) ← g(N) + k(N, M)
17:             f(M) ← g(M) + h(M)
18:             if M ∈ OPEN then
19:                 continue
20:             end if
21:             if M ∈ CLOSED then
22:                 Propagate-Improvement(M)
23:             else
24:                 add M to OPEN
25:             end if
26:         end if
27:     end for
28: end whilereturn empty list
```

- Propagate improvement simply does line 13 to 17 recursively.
- A* is admissible if

1. The branching factor is finite, otherwise you cannot even generate the neighbors.
2. Every edge has a cost greater than a small constant ϵ, however it is possible to get stuck in an infinite path with a finite cost.
3. For all nodes $h(n) \leq h*(n)$

- If a path exists to the goal node, then the OPEN list always contains a node n' from an optimal path. Moreover, the f value of that node is not greater than the optimal value.