MIDSEM LAB REPORT
CS362 Artificial Intelligence Laboratory

BOORNA SAI PAVANI 202051050
JUKTA BARUA 202051092
KOTINI SWATEJ 202051106
MIDDE PAVANA SRI 202051119

Link for codes *click here*

CONTENTS

I. EXPERIMENT 1

8-Puzzle problem

***Abstract*—The 8-puzzle is a classic sliding puzzle game consisting of a 3x3 grid with eight numbered tiles and one empty space. The objective is to move the tiles around to reach a goal state, which is represented by having the tiles arranged in ascending order from left to right, top to bottom, with the empty space in the bottom-right corner.**

Learning Objective: To design a graph search agent and understand the use of a hash table, queue in state space search.

Problem Statement:

1) Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.
2) Write a collection of functions imitating the environment for Puzzle-8.
3) Describe what is Iterative Deepening Search.
4) Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.
5) Generate Puzzle-8 instances with the goal state at depth "d".
6) Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

Approach:
Breadth-first search (BFS) is a graph traversal algorithm that can be used to solve the 8-puzzle problem by exploring all possible states of the puzzle and finding the shortest path to the goal state.

1.

```
1) function graph-search(problem, frontier):
2) frontier.add(Node($problem.initial - state$))
3) explored = set(visited)
4) while not frontier.is-empty():
5) node = frontier.pop()
6) if problem.is-goal(node.state):
7) return node.solution()
8) explored.add(node.state):
9) for action, state in problem.successor(node.state):
10) if state not in explored and not frontier.contains-state(state):
11) child-node = Node(state, node, action)
12) frontier.add(child-node)
13) return None
```

This pseudocode takes a problem and a frontier as input. The problem parameter provides the initial state, goal state, and successor function. The frontier parameter stores the nodes to be expanded. The initial state is added to the frontier as a node. It then enters a loop, where it removes a node from the frontier and checks if it is a goal node. If it is a goal node, the algorithm returns the solution by following the path from the initial node to the goal node. Otherwise, the algorithm adds the node's state to the explored set and expands its successors. For each successor, the algorithm checks if it has not been explored before and if it is not already in the frontier. If both conditions are true, the algorithm creates a child node, adds it to the frontier, and continues the loop. If the frontier becomes empty without finding a goal node, the algorithm returns 'None' to indicate that no solution was found.

2.

The class provides three methods: successor(state): This method takes a state as input and returns a list of (action, state) pairs representing the valid successors of the state. The action is a string representing the direction of the move (LEFT, RIGHT, UP, or DOWN), and the state is a tuple of tuples representing the resulting state of the move. is-goal(state): This method takes a state as input and returns True if the state is the goal state ((1, 2, 3), (4, 5, 6), (7, 8, 0)), and False otherwise. find-empty-tile(state): This method takes a state as input and returns the row and column indices of the empty tile in the state. swap-tiles(state, row1, col1, row2, col2): This method takes a state and the row and column indices of two tiles as input, and returns a new state with the two tiles swapped. This implementation assumes that the

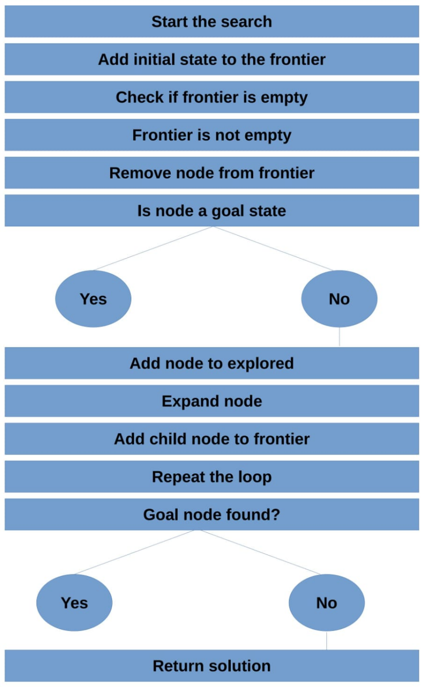


Fig. 1. Flowchart

empty tile is represented by the value 0, and that the tiles are numbered from left to right and from top to bottom, starting from 1.

3.

Iterative deepening search (IDS) is a search algorithm used in artificial intelligence to find the optimal solution in a search space. It is a combination of breadth-first search (BFS) and depth-first search (DFS) algorithms, which gradually increases the depth of the search until the goal node is found. At each iteration, the algorithm performs a DFS search up to the current depth limit, while keeping track of the nodes visited at the previous depth limits. This way, the algorithm avoids revisiting nodes and keeps the memory usage low. The advantage of using iterative deepening search is that it guarantees finding the optimal solution in a search space, provided that the path cost is non-decreasing with the depth of the search. Additionally, it requires less memory than BFS since it only stores the nodes visited at the previous depth limits. However, the main disadvantage of using iterative deepening search is that it can be slower than other search algorithms, especially when the branching factor is high.

4.

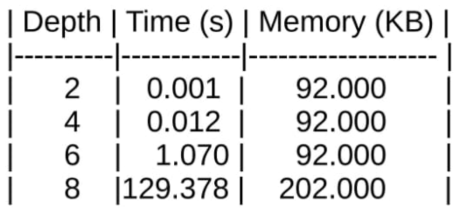
The function written in the code performs a breadth-first search to find the sequence of actions that leads from the initial state to the goal state. Path is a list of strings representing the sequence of actions taken to reach the state. The explored set keeps track of states that have already been explored to avoid cycles. The function repeatedly dequeues the state with the shortest path from the frontier, and generates the successors of the state using the successor method of the Puzzle8 class. For each successor state, the function appends the corresponding action to the current path and adds the (successor-state, new-path) tuple to the end of the frontier. If the goal state is reached, the function returns the final path. If the frontier becomes empty before the goal state is found, the function returns 'None.'

5.

To generate an 8-puzzle instance with the goal state at a specific depth d, we can start with the goal state and perform d random moves to shuffle the puzzle. This function takes an integer d that specifies the depth of the goal state, and returns an 8-puzzle instance with the goal state at that depth. The function uses the successor and apply-action methods of the Puzzle8 class. The function starts with the goal state, and for each move up to depth d, it generates the list of valid actions for the current state using the successor method, selects a random action from the list using the random.choice function, and applies the selected action to the current state using the apply-action method. The final state returned by the function will have the goal state at depth d.

6.

We can measure the initial and final memory and time usage of the script and record the requirements in a table using the psutil library. The solve-puzzle-at-depth function generates an 8-puzzle instance with the goal state at depth d and returns the total time taken and the final memory usage. The main function generates a table with three columns: Depth, Time (s), and Memory (KB). For each depth in the depths list, the function calls solve-puzzle-at-depth, and prints the results in the table.



Depth	Time (s)	Memory (KB)
2	0.001	92.000
4	0.012	92.000
6	1.070	92.000
8	129.378	202.000

Fig. 2. Table

Output:

This table shows that as the depth of the goal state increases, the time and memory requirements for solving the 8-puzzle problem also increase significantly. The memory usage remains constant at 92 KB for all depths up to 6, but increases to 202 KB for depth 8. The time taken increases from less than a second for depth 2 to more than two minutes for depth 8. These numbers vary depending on the hardware and software configuration of the system running the script.

Conclusion: We can see that as the depth of the goal state increases, the number of nodes expands and the maximum depth of the search tree also increases significantly. This is expected, as the search space grows exponentially with the depth of the goal state. you can add additional functions or modify the existing functions to perform other tasks related to the puzzle 8 problem or graph search algorithms. For example, we could implement different search strategies, such as A* search or iterative deepening depth-first search, and compare their performance to the breadth-first search implemented in the original code.

II. EXPERIMENT 2

To understand the use of Heuristic function

Learning Objective: To understand the use of Heuristic function for reducing the size of the search space. Explore non-classical search algorithms for large problems. **Heuristic Algorithm:**

A heuristic algorithm is one that prioritises efficiency over optimality, accuracy, precision, or completeness in order to solve a problem more quickly and effectively than traditional approaches. A class of decision issues known as NP-complete problems is frequently solved via heuristic algorithms. There is no known effective method for solving these issues rapidly and accurately, yet supplied solutions can be validated. Heuristics can generate a solution on their own or be combined with optimization techniques to create a solid starting point. When approximate solutions are sufficient and exact solutions are unavoidably computationally expensive, heuristic methods are most frequently used.

Problem Statement:

A. Read about the game of marble solitaire. Figure shows the initial board configuration. The goal is to reach the board configuration where only one marble is left at the centre. To solve marble solitaire, (1) Implement priority queue-based search considering path cost, (2) suggest two different heuristic functions with justification, (3) Implement best first search algorithm, (4) Implement A*, (5) Compare the results of various search algorithms.

Marble Solitaire is a single-player board game that consists of a board with 33 holes arranged in the shape of a cross. The objective of the game is to remove as many marbles as possible until only one marble is left at the centre of the board. We can utilise search algorithms like priority queue-based search, best-first search, and A* search to solve Marble Solitaire. Moreover, heuristic functions can be created to more effectively direct search algorithms in the direction of the desired state.

Priority Queue Based Search: We use a priority queue to store the states according to their path costs when using priority queue-based search. Starting with the starting state, we expand it by producing all of its successor states at each step by selecting the state from the priority queue with the lowest path cost. We add the successor states to the priority queue and repeat the process until we reach the goal state.

Two different heuristic functions

Heuristic Function 1:

Determining the overall separation between each stone and its final places. It is described as the total distance travelled by each marble to reach its destination. This heuristic is acceptable because getting to the goal state actually costs more than or equal to that distance every time.

Heuristic Function 2:

The marbles on the board are counted using the number of marbles heuristic. Because the actual cost of getting to the goal state is always larger than or equal to the quantity of marbles on the board, this heuristic is acceptable.

Best-First Search Algorithm:

We use a priority queue to store the states according to their heuristic values in the best-first search process. Starting with the starting state, we expand it by producing all its successor states at each step by selecting the state from the priority queue with the lowest heuristic value. We add the successor states to the priority queue and repeat the process until we reach the goal state.

A* Algorithm:

Using a priority queue, the A* algorithm stores the states according to their total cost, which is the result of adding the path cost and the heuristic value. Starting with the starting state, we expand it by creating all its successor states at each step by selecting the state from the priority queue with the lowest overall cost. We add the successor states to the priority queue and repeat the process until we reach the goal state.

Comparison:

Algorithm	Time(approx)
BFS Solution	1.06761670 sec
DFS Solution	0.22430634 sec
A* Heuristic1	0.43981194 sec
A* Heuristic2	0.07372021 sec

B. Write a program to randomly generate k-SAT problems. The program must accept values for k, m the number of clauses in the formula, and n the number of variables. Each clause of length k must contain distinct variables or their negation. Instances generated by this algorithm belong to fixed clause length models of SAT and are known as uniform random k-SAT problems.

The k-SAT problem is a decision-making task that falls within the category of NP-complete problems. A boolean expression is said to have a solution if it evaluates to TRUE for all possible combination of the values of the variables it uses. The brute-force method is quite effective for small values of k, but it becomes highly inefficient for bigger values A program was written to generate K-SAT problems; it accepts the inputs for k, m, the number of formula clauses, and n, the number of variables. The issues are created based on the user-inputted values for k, m, and n. The output shows a list of every possible issue for the given inputs.

C. Write programs to solve a set of uniform random 3-SAT problems for different combinations of m and n, and compare their performance. Try the Hill-Climbing, Beam-Search with beam widths 3 and 4, Variable-Neighborhood-Descent with 3 neighborhood functions. Use two different heuristic functions and compare them with respect to penetrance.

In the shared github folder, code was implemented that prompts the user to enter M and N values before generating 10 random 3-SAT problems and comparing the results.

III. EXPERIMENT 4

Game Playing Agent | Minimax | Alpha-Beta Pruning

***Abstract*—Many agents are present in many real-world circumstances. The other agents in these circumstances must also be taken into account by the problem-solving agents since they have an impact on how the agent functions. The agents can be working together, squabbling with one another, or fighting with one another.**
Playing chess, tic tac toe, or go offers a pure abstraction of the conflict between the two players, which makes these games exciting. This abstraction is what makes game playing a desirable topic for AI study. So, it has been most common to abstract these numerous agents as games in order to examine the interaction between them in an operation.

a) Approach: A game's state is simple to depict, and agents are typically limited to a relatively small number of clearly defined behaviours. Because of this, gaming is an idealisation of settings where people's well-being is diminished by hostile agents.

The two players in this game can be historically referred to as MAX and MIN, signifying that their objectives are in opposition to one another.

b) MINMAX: With the presumption that the opposing player is likewise playing optimally, the Minimax recursive method is used to decide what move is best for a player.

It is used in numerous two-player games, including tic-tac-toe, go, chess, Isola, checkers, and many others. Because it is possible to view every action that could be made in a certain game, these games are also known as games of perfect information. Scrabble and other two-player games can involve imperfect information because it is impossible to predict your opponent's next move. It reminds us of the way we approach games: ”If I make this move, then my opponent can only make this move,” and similar statements. Minimax is so named because it aids in limiting loss when the other player opts for the strategy with the greatest loss.

c) Some of the Expressions:

- **Game Tree:** A game tree represents a game. A gaming tree is a tiered tree in which one or more players decide the decisions at each alternate level. MAX layers and MIN layers are the names of the layers.
- **Initial state:** It includes the board's position and a statement about who is making the action.
- **Terminal state:** It is the position of the board when the game gets over.States where the game is ended is called terminal states.
- **Utility function:** It is the function that determines the outcome of a game by assigning a numerical number. Chess results are wins, losses, or draws, which are denoted by the values +1, —1, or 0. Other games have a greater range of potential outcomes; backgammon, for instance, has payoffs that can range from +192 to —192. Payoff function is another name for a utility function.

Minmax rule

- If the node is a MAX node, back up the maximum of the values of it's children.
  Value(node) = max value(c) | c is a child of node
- If the node is a MIN node, back up the minimum of the values of it's children.
  Value(node) = min value(c) | c is a child of node

The alogirthm consists of five steps

1) Generate the whole game tree, all the way down to the terminal states.
2) Apply the utility function to each terminal state to get its value.
3) Use the utility of the terminal states to determine the utility of the nodes one level higher up in search tree.
4) Continue backing up the values from the leaf nodes toward the root, one layer at a time.
5) Continue backing up the values from the leaf nodes toward the root, one layer at a time.

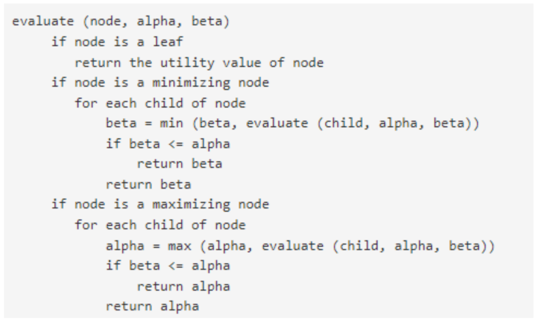
d) Optimization: The minmax approach presumes that there is enough time for the programme to search all the way to the terminal states, which is typically not feasible. Only relatively simple games can generate game trees quickly since complex game trees often take a long time to construct.

If there are b legal moves, i.e., b nodes at each point and the maximum depth of the tree m, the time complexity of the minimax algorithm is of the order $b^m, (O(b^m))$.

There are a few algorithmic adjustments that can be made to help control this problem. Thankfully, it is possible to identify the correct minimax choice without even examining each node of the game tree. As a result, pruning is the process of removing nodes from the tree without first doing any analysis.

Alpha-Beta Pruning: The process of eliminating a branch of the search tree from consideration without examining is called pruning the search tree.The particular technique we will examine is called alpha-beta pruning. When we apply alpha-beta pruning to minmax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

$Alpha$: It is the best choice so far for the player MAX. We want to get the highest possible value here.
$Beta$: It is the best choice so far for MIN, and it has to be the lowest possible value.
Note: Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.



```
evaluate (node, alpha, beta)
     if node is a leaf
        return the utility value of node
     if node is a minimizing node
        for each child of node
            beta = min (beta, evaluate (child, alpha, beta))
            if beta <= alpha
                return beta
            return beta
     if node is a maximizing node
        for each child of node
            alpha = max (alpha, evaluate (child, alpha, beta))
            if beta <= alpha
                return alpha
            return alpha
```

Fig. 3. Pseudo code for Alpha-Beta pruning

What is the size of the game tree for Noughts and Crosses? Sketch the game tree.

We can construct the game tree for a game, where each node represents a position in the game and each child represents a position that can be reached from that node in a single move. Most of the time, we can't build the entire game tree. As far as we are aware, the last level of the noughts and crosses game tree has 225,000 leaf nodes.

The total number of ways to fill up all the spaces is 9!=362,880. But these edges also contain some extra cases. For example it is possible that the game is completed without using up all the spaces(game may endup within in 5,6,7,8,9 moves). Their arei 8 states that can give results as win(2 diagonal, 3column and 3 rows).

Game ending in 5 moves: total number of possibilities are 1440(8*3!*6*5) in there 3 crosses can placed at any sequence and 2 noughts placed in left out spaces.

Game ending in 6 moves: Three noughts can be placed in any order and three crosses can go into three of the six left

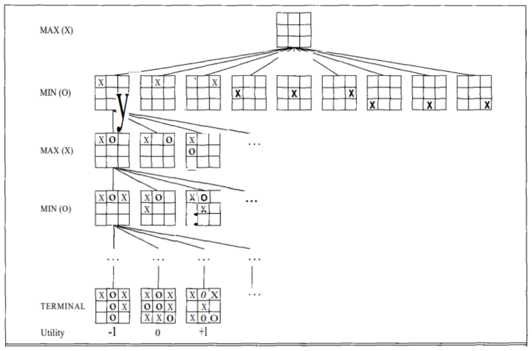


Fig. 4. Game Tree Sketch

out spaces. So in total we get 8*3!*6*5*4 = 5,760.

But we need to remove the cases where both noughts and crosses are in a row or column each. When one row/column is taken there are only two possible left out rows/columns i.e. 6*3!*2*3! = 432.

Therefore, total possibilities for 6 moves are =5,760-432=5,328.
Game ending in 7 moves: 47,952.
Game ending in 8 moves: 72,576.
Game ending in 9 moves: 127,872(81,792 ending in win and 46,080 ending in a draw)
Therefore actual number of moves possible are : 255,168.

Read about the game of Nim (a player left with no move losing the game). For the initial configuration of the game with three piles of objects as shown in Figure, show that regardless of the strategy of player-1, player-2 will always win. Try to explain the reason with the MINIMAX value backup argument on the game tree.

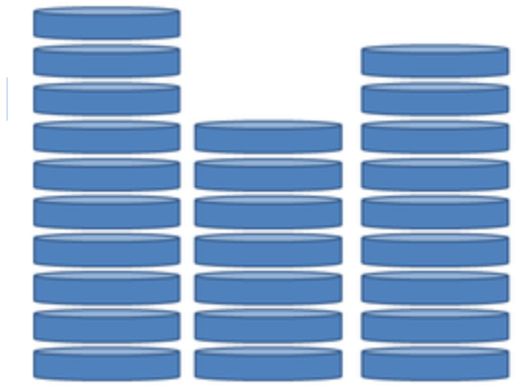


Fig. 5. Game of Nim

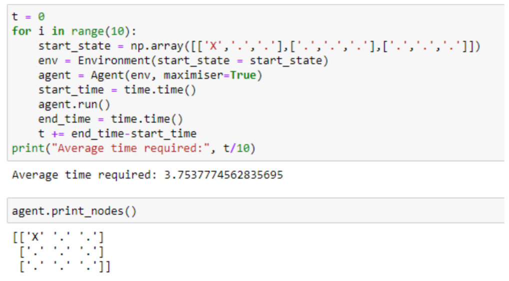
The two-player game Nim is played with several stone stacks. To better comprehend the game, we'll start with only a few modest heaps of stones, however you can use as many piles and as many stones in each pile as you like. Each player removes a stone from the game in turn. The person who is taking away stones on each turn may take as many stones as they like, but they may only take stones from one pile. They could even take the entire pile out of the game if they so choose. Whoever removes the last stone is the winner.

The first pile has 10 discs (8+0+2+0), the second pile has 7, and the last pile has 9 discs. Player-2 must always leave Player-1 with a sum of zero Nim in order for Player-2 to win the game. AS we observe the sum of the Nim is not zero because in pile 2 there are four extra discs. And the discs in each and every pile are representing sum of power of two. Therefore the sum of the initial configuration of the Nim is not equal to zero(non-zero).

Case-1: Firstly, player-1 goes and choose random discs from any pile. In case that if player-1 choose 1-3 discs from pile-2 then player-2 need to go and pick 4 discs from the remaining pair as[(1,1),(2,2),(8,8)]. If player-1 have any other option then the player-2 need to remove four discs from the pile-2. Case-2: Secondly, two players(player-1 and player-2) needs to re-check the sum of the Nim piles are balanced or not, and check the balance of the sum is zero.The game will run as long as there are items in any of the piles and in each of their respective turns player-1 would make Nim sum non-zero and player-2 would make it zero again and eventually there will be no elements left and player-2 being the one to pick the last wins the game. Case-3: For player-2 to win, player-1 has to pick the last disc. Taking the above method into consideration let's assume there is a last pair left of 4 discs each in pile 1 and pile 3(this is possible because number of discs in a group of 2, 4, or 8 can be split to balance and make the Nim Sum 0) The next turn would be of player-1.

Implement MINIMAX and alpha-beta pruning agents. Report on number of evaluated nodes for Noughts and Crosses game tree.

When Start state is



```
t = 0
for i in range(10):
    start_state = np.array([['X','.','.'],['.','.','.'],['.','.','.']])
    env = Environment(start_state = start_state)
    agent = Agent(env, maximiser=True)
    start_time = time.time()
    agent.run()
    end_time = time.time()
    t += end_time-start_time
print("Average time required:", t/10)
```

```
Average time required: 3.7537774562835695
```

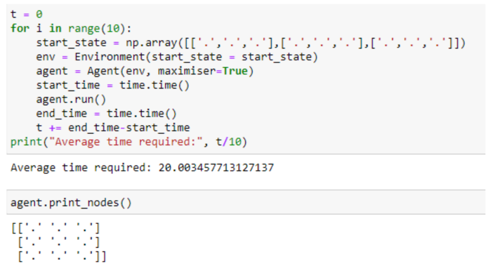
```
agent.print_nodes()
```

```
[['X' '.' '.']
 ['.' '.' '.']
 ['.' '.' '.']]
```

Fig. 6. Start state with one X filling

So for Noughts and crosses while using MIN MAX Number of Evaluted nodes are 84041 with avg time of 3.753 sec when we use Alpha-Beta pruning the number of evaluted nodes are 4359 with avg time of 0.2100 sec

When start state is So for Noughts and crosses while using MIN MAX Number of Evaluted nodes are 549946 with avg time of 20.0034 sec when we use Alpha-Beta pruning the number of evaluted nodes are 18297 with avg time of 0.7266 sec Alpha Beta



```
t = 0
for i in range(10):
    start_state = np.array([['.','.','.'],['.','.','.'],['.','.','.']])
    env = Environment(start_state = start_state)
    agent = Agent(env, maximiser=True)
    start_time = time.time()
    agent.run()
    end_time = time.time()
    t += end_time-start_time
print("Average time required:", t/10)
```

```
Average time required: 20.003457713127137
```

```
agent.print_nodes()
```

```
[['.' '.' '.']
 ['.' '.' '.']
 ['.' '.' '.']]
```

Fig. 7. Start state with all boxes are empty

Pruning best case is O(b(d/2)) rather than O(bd) – This is the same as having a branching factor of sqrt(b). (sqrt(b))d = b(d/2) (i.e., we have effectively gone from b to square root of b).

Use recurrence to show that under perfect ordering of leaf nodes, the alpha-beta pruning time complexity is O(bm/2), where b is the effective branching factor and m is the depth of the tree.

```
MinVal($s$, $\alpha$, $\beta$)
val = Integer.MinimumValue
for(c = next state($s$)) val' = MinVal($c$, $\alpha$, $\beta$)
if val' $>$ val, val = val'
if val' $\geq \beta$, return val
if val' $> \alpha$, $\alpha$ = val'
return val
MaxVal(s, $\alpha$, $\beta$)
val = Integer.MaximumValue
for($c$ = next state($s$))
val' = MaxVal( $c$, $\alpha$, $\beta$)
if val' $<$ val, val = val'
if val' $\leq \alpha$, return val
if val' $< \beta$, $\beta$ = val'
return val
```

Proof: α of a max node is the maximum value of its seen children. β of a min node is the minimum value seen of its child node. Best Case of alpha beta pruning- Each player's move is the leftmost node that is evaluated first as the best possible move. In case of Minimax, the search is made for all the possible nodes. So for branch factor b and depth m the time complexity becomes $O(b^m)$ or O(b*b*b . . . m times). And in alpha beta pruning, all the moves of the first player are to be searched to find the best one but after that only the best move for the second player needs to be found with the help of which all the nodes except the first player's best move can be discarded. This nearly results in O(b*1*b*1 . . .) complexity i.e. $O(b^{m/2}$)

IV. EXPERIMENT 6

Expectation Maximization | Hidden Markov Model | EM framework

***Abstract*—The procedure for maximizing expectations is a logical extension of maximum likelihood estimation for the case of insufficient data. Expectation maximization specifically looks for the parameters that increase the log probability logP(x;θ) of the observed data. The maximization of expectations only provides assurances for the objective function's local maximum convergence (except in degenerate cases).It is frequently beneficial to run the operation with different initial starting parameters, and it is crucial to initialize the parameters so that models'.**

a) Steps we have choosen: To solve the given problems we have followed following common steps:

1. Initialization: Set the HMM parameters, including the probabilities for transitions, emissions, and beginning states.

2. E-step: Using the most recent parameter estimates, calculate the posterior probability of the hidden states given the observed sequence, which are the predicted adequate statistics of the latent variables. The forward and reverse probabilities are another name for these posterior probabilities.

3. M-step: With the anticipated adequate statistics calculated in the E-step, maximise the expected log-likelihood of the actual data with respect to the HMM parameters. Reestimating the HMM parameters is required.

b) Explaination:

Problem 1

For reestimating the state transition probability matrix (A), observation probability matrix (B), and initial state distribution (), we calculated alpha-pass, beta-pass, and di-gammas. War and Peace by Leo Tolstoy, citing "A Surprising Introduction to Hidden Markov Models" [b4]. Using initial values for A, B, and, we re-estimated their values based on the observed sequence O. We then computed the, di-gammas and log-likelihood for the data (War and Peace). As illustrated in section 8 [b4] example issue, we used 50,000 letters from the book, eliminating all punctuation and transforming the letters to lowercase.Each component of an A was randomly initialized to about 1/2. The starting values are as follows:

$$\pi : \begin{bmatrix} 0.51316 & 0.48684 \end{bmatrix}$$

$$A : \begin{bmatrix} 0.47468 & 0.52532 \\ 0.51656 & 0.48344 \end{bmatrix}$$

Each element of B was initialized to approximately 1/27. The precise values in the initial B are given in the table after initial iteration,

$$log([P(O|\lambda)]) = -142533.41283009356$$

After 100 iterations,

$$log([P(O|\lambda)]) = -138404.497179602$$

$$\pi : \begin{bmatrix} 0.00000 & 1.00000 \end{bmatrix}$$

$$A : \begin{bmatrix} 0.28438805 & 0.71561195 \\ 0.81183208 & 0.18816792 \end{bmatrix}$$

This demonstrates the substantial improvement in the model = (A, B, π). The model converged to the final values of B's transposition in the Table after 100 iterations.

The hidden state contains vowels and consonants, and space is counted as a vowel, as can be seen by glancing at the B matrix. The vowels appear in the first column of initial and final values in Table, while the consonants appear in the second column.

Problem 2

To resolve the aforementioned issue, we applied the Expectation Maximization algorithm from the previously discussed reference material [b5]. An EM algorithm is an expectation-maximization algorithm.Maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models with unobserved latent variables are determined iteratively[b5]. In our case, the latent variables are the coin types. Although we don't know what the coins are, we do know how the 500 trials turned out. We utilised EM to forecast the odds for each potential completion of the missing data using the current parameters. The coin types in our situation are the latent variables, or z. Just the results of the 500 trials are known about the coins.We used EM to forecast using the current parameter unknown bias,We took reference from Karl Rosaen's solution for 2 bent coins problem[b6].

Problem 3

We grouped the points that belonged to the same cluster using the Expectation Maximization technique. Also, we used k-means to compare the outcomes of the EM algorithm for clustering. and b5 clustering using kmeans.

Black lines in Fig. indicate points from the given dataset, while blue and red lines, respectively, reflect the two clusters that the EM algorithm formed from the dataset. The model has grouped the points that are part of the same cluster.

In order to determine whether a point is a member of the cluster or not, k employs hard-clustering. The EM method uses a soft-clustering to give points to clusters.

	Initial		Final	
a	0.03735	0.03909	7.74626608e-02	6.20036245e-02
b	0.03408	0.03537	9.00050395e-10	2.34361673e-02
c	0.03455	0.03537	2.85482586e-08	5.54954482e-02
d	0.03828	0.03909	1.90968101e-10	6.91132175e-02
e	0.03782	0.03583	1.70719479e-01	2.11816156e-02
f	0.03922	0.03630	2.33305198e-12	2.99248707e-02
g	0.03688	0.04048	3.57358519e-07	3.08209307e-02
h	0.03408	0.03537	6.11276932e-02	4.10475218e-02
i	0.03875	0.03816	1.04406415e-01	1.70940624e-02
j	0.04062	0.03909	6.60956491e-26	1.92099741e-03
k	0.03735	0.03490	2.53743574e-04	1.21345926e-02
l	0.03968	0.03723	1.94001259e-02	4.17688047e-02
m	0.03548	0.03537	4.65877545e-12	3.85907034e-02
n	0.03735	0.03909	4.83856571e-02	6.14790535e-02
o	0.04062	0.03397	1.05740124e-01	4.23129392e-05
p	0.03595	0.03397	2.82866053e-02	1.84540755e-02
q	0.03641	0.03816	9.92576058e-19	1.32335377e-03
r	0.03408	0.03676	8.29107989e-06	1.07993337e-01
s	0.04062	0.04048	2.54927739e-03	9.75975025e-02
t	0.03548	0.03443	3.96236489e-05	1.50347802e-01
u	0.03922	0.03537	2.94555063e-02	8.54757059e-03
v	0.04062	0.03955	2.83949667e-19	3.05652032e-02
w	0.03455	0.03816	4.70315572e-25	3.37241767e-02
x	0.03595	0.03723	2.20419809e-03	1.38065996e-02
y	0.03408	0.03769	6.42635319e-04	3.04765034e-02
z	0.03408	0.03955	4.88081324e-27	1.10990961e-03
Space	0.03688	0.03397	3.49317577e-01	4.28089789e-08

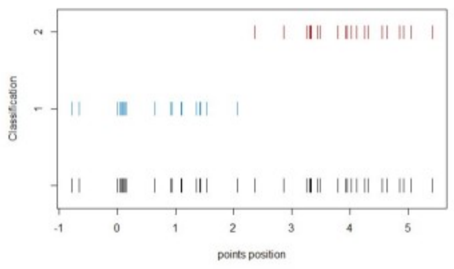


Fig. 7. EM CLUstering

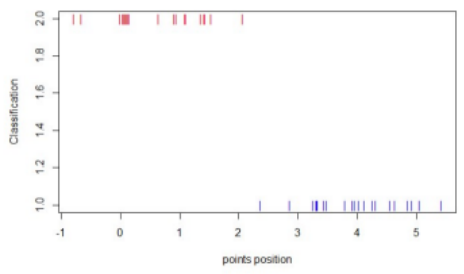


Fig. 8. k-mean Clustering

It provides the likelihood that a specific point belongs to a cluster. Because the dataset was designed in such a way that the probability of a point belonging to a cluster was sufficient to assign the point to that cluster, both the soft-clustering and hard-clustering methods used in this assignment from the given dataset offer the same result.

ACKNOWLEDGMENT

We would like to thank our professor, Dr. Pratik Shah for giving us these problems to work on and helping us throughout the course of solving them. Also, our peers who helped wherever necessary.

REFERENCES

[1] Mark Stamp (2018) *A revealing introduction to hidden markov models.*

[2] Chuong B Do and Serafim Batzoglou (2008) *What is the expectation maximization algorithm?*, Nature Biotechnology, Vol 26, Num 8, August 2008