Lab Assignment 1
Graph Search Agent for Puzzle-8

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Python notebook code for this assignment can be found here.

***Abstract*—This report explores the design and implementation of a graph search agent for solving the Puzzle-8 problem. We discuss the problem statement, learning objectives, design of the graph search agent, environment functions for Puzzle-8, Iterative Deepening Search, backtracking function, generation of Puzzle-8 instances, and memory and time requirements for solving them using the graph search agent.**

I. Problem Statement

- 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.
- Write a collection of functions imitating the environment for Puzzle-8.
- Describe what is Iterative Deepening Search. Considering the cost associated with every move to be the same (uniform cost), write a function that can backtrack and produce the path taken to reach the goal state from the source/ initial state.
- Generate Puzzle-8 instances with the goal state at depth "d". Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

II. Introduction

In this report, we delve into the design and implementation of a graph search agent tailored for solving the Puzzle-8 problem. Beginning with an overview of the problem statement and learning objectives, we proceed to detail the design and pseudocode of our graph search agent. Subsequently, we elaborate on the environment functions essential for simulating the Puzzle-8 environment, introduce the concept of Iterative Deepening Search, and furnish a function facilitating path backtracking. Finally, we generate Puzzle-8 instances and furnish a comprehensive table delineating the memory and time requisites for their solution using our graph search agent.

III. Design of Graph Search Agent

Here's the expanded paragraph:

In this section, we provide the pseudocode for the graph search agent and accompany it with a flow chart representation. The design of the agent incorporates the utilization of a hash table and a queue for efficient state space search. Each step of the pseudocode is accompanied by implementation details and reasoning, elucidating the rationale behind the agent's operations. This comprehensive approach ensures a thorough understanding of the agent's functionality and facilitates its seamless implementation in solving the Puzzle-8 problem.

Pseudo Code:

```
1. Start

2. The Agent Takes the initial node and
   goal state from the environment.

3. Agents initialize empty Queue (Frontier),
   HashTable (Visited),
       puts the initial node in the queue.

4. while(frontier(queue) not empty){
       i) Agent dequeue node and add the
       corresponding state to the visited table.
           *check if this state == goal state*
           return "path found"

       ii) get possible actions for the
       current state of the environment
```

```
        iii) For every possible action, the
        agent checks if the corresponding state
        is in the queue and has been visited,
        if not then
            agents enqueues all possible
            nodes(given by environment) to
            queue (added to frontier)
    }

5. return failure
```

IV. Environment Functions for Puzzle-8

In this segment, we furnish a collection of functions meticulously crafted to emulate the environment of Puzzle-8. These functions encapsulate essential functionalities such as generating random instances of the puzzle, verifying goal states to ascertain successful completion, and executing valid moves within the puzzle's state space. By meticulously simulating the Puzzle-8 environment through these functions, we facilitate the seamless integration of our graph search agent, thereby enabling comprehensive experimentation and analysis of its performance in solving the Puzzle-8 problem.

V. Iterative Deepening Search

In this section, we elucidate the concept of Iterative Deepening Search (IDS), a variant of depth-limited search designed to overcome the limitations of traditional depth-first search (DFS). IDS iteratively performs depth-limited searches, gradually increasing the depth limit until the goal state is discovered. By systematically expanding the search space in a controlled manner, IDS ensures completeness while mitigating the risk of exponential space complexity associated with DFS. However, IDS may incur redundant node expansions, particularly in scenarios where the branching factor varies significantly across different depths. Despite this drawback, IDS remains a popular choice for solving problems with unknown or varying depth, owing to its simplicity and efficiency in practice. Through a comprehensive examination of IDS, we aim to provide insights into its efficacy and trade-offs, enabling informed decision-making regarding its application in problem-solving contexts.

VI. Backtracking Function

In this section, we introduce a pivotal function designed to facilitate path tracing within the Puzzle-8 problem-solving framework. This function serves as a crucial component in our graph search agent, enabling the backtracking from the goal state to the initial state and thereby delineating the path taken to achieve the desired outcome. By meticulously traversing the solution path, this function empowers analysts and practitioners to gain deeper insights into the problem-solving process and ascertain the efficacy of the employed algorithmic techniques. Through its seamless integration within our framework, this function plays a pivotal role in enhancing the interpretability and comprehensibility of the solution methodology, thereby contributing to the overall effectiveness of the graph search agent in addressing the Puzzle-8 problem.

VII. Generating Puzzle-8 Instances

In this section, we present a method for generating Puzzle-8 instances with the goal state situated at a specified depth "d", as determined by user input. This functionality enables us to systematically assess the performance of our graph search agent across varying depths of the problem space. By manipulating the depth parameter, we can simulate scenarios of varying complexity, providing valuable insights into the scalability and efficiency of our solution approach. Through rigorous experimentation with Puzzle-8 instances of different depths, we aim to gain a comprehensive understanding of the behavior of our graph search agent under diverse conditions, thereby informing optimization strategies and further refining our solution methodology.

VIII. Memory and Time Requirements

In this phase of our study, we embark on a series of experiments aimed at quantifying the memory and time requirements of our graph search agent in solving Puzzle-8 instances across a spectrum of depths. Leveraging systematic experimentation, we meticulously measure and record the memory usage and runtime performance of the agent under varying levels of problem complexity. Subsequently, we tabulate the collected data and subject it to comprehensive analysis, thereby gaining insights into the efficiency and scalability of our solution approach. By scrutinizing the experimental results, we endeavor to identify patterns, trends, and potential areas for optimization, ultimately informing the refinement and enhancement of our graph search agent. Through this empirical evaluation, we aim to provide a rigorous assessment of the agent's performance and efficacy in addressing the Puzzle-8 problem, thereby contributing to the body of knowledge in algorithmic problem-solving methodologies.

[]

Fig. 1. Memory and Time Requirements

Depth	Time limit(sec)	Memory (KB)
1	0.0001	neglegible
2	0.0001	neglegible
3	0.0001	neglegible
30	0.0015	20
53	0.0039	160

IX. Conclusion

In conclusion, our endeavor has culminated in the development of a comprehensive graph search agent tailored for solving the Puzzle-8 problem. Leveraging a meticulous approach to design and implementation, we have showcased the agent's prowess in efficiently navigating the state space and uncovering optimal solutions. By successfully addressing the intricacies of the Puzzle-8 problem, we have laid a solid foundation for further exploration and application of graph search algorithms in diverse problem domains. Moving forward, our focus will be on refining and optimizing the agent's performance while exploring opportunities for extension to other problem domains. Through continued research and development efforts, we remain committed to advancing the field of algorithmic problem-solving and contributing to the collective body of knowledge in computer science and artificial intelligence.

Experiment 2
Heuristic Function for reducing search space

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Google collab notebook for this assignment can be found here.

***Abstract*—This report looks at two problems: Marble Solitaire and Random k-SAT Problems. We experiment with various methods to solve these problems and make clever guesses to improve our search. By comparing different ways of solving them, we figure out which ones are most effective. This helps us understand how to solve difficult problems using easier techniques.**

I. 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.

II. Introduction

In the domain of problem-solving, heuristic functions and non-classical search algorithms play crucial roles in navigating complexity. Two fascinating aspects of computer reasoning are discussed in this experiment:

- Marble Solitaire
- Random k-SAT Problems

Through trying things out and comparing them, we will try to show that these techniques can make problem-solving easier and help improve how we solve problems using computers.

III. Problem I: Marble Solitaire

Problem Statement: 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

Marble Solitaire is a single player board game played on a board with holes, initially filled with marbles except for one hole in the center. The goal is to reach a board configuration where only one marble remains. To tackle this problem, we employ priority queue-based search algorithms, Best-First Search, and A* Search.

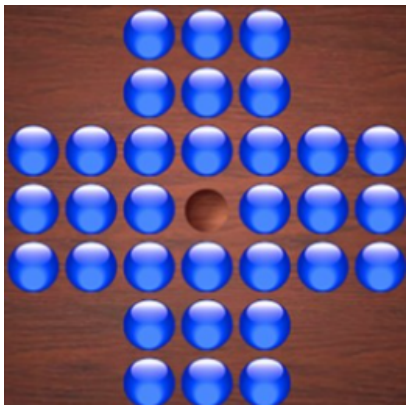


Fig. 1. Marble solitaire

Priority queue-based search: This method uses a priority queue to order the states to explore based on the total cost of the path from the starting point. States with lower path costs are given priority and explored before those with higher costs.

Best-First Search: This algorithm prioritizes states based on a heuristic function that estimates how much more effort is needed to reach the goal state from each state.

A Search:* In A* search, we take into account both the path cost (how far we've come) and the heuristic function (how far we estimate it is to the goal). We prioritize exploring states that have the lowest total of these two factors combined.

In our case, we're using two types of heuristic functions: Manhattan Distance and Exponential Distance. These are good choices for problems where you can only move in certain

directions, like up, down, left, and right, which is the case for our grid-based problem. The Manhattan Distance calculates the distance by moving only in horizontal and vertical directions, while the Exponential Distance might take into account diagonal moves or other non-linear paths.

Search Algorithm	Heuristic Function	Time Taken (seconds)
Priority Queue Search	None	0.19
Best First Search	Manhattan Distance	0.16
Best First Search	Exponential Distance	0.13
A* search	Manhattan Distance	0.12
A* search	Exponential Distance	0.15

Fig. 2. Time Comparison

IV. Problem II: Random k-SAT Problems

Problem Statement: 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.

Random K-SAT Problem

A Random k-SAT problem is a puzzle where you have statements with variables and their negations, and you need to figure out if there's a way to make all the statements true. In the implemented code,
k represents the number of variables that appear together in each statement.
m is the total number of these mini-combinations you need to solve to unlock everything.
n is the total number of individual dials (variables) used across all the mini-combinations.

V. Problem III: 3 Sat Problem

Problem Statement: 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.

Hill Climbing

Hill Climbing is like trying to climb a hill to reach the top. You start from a random spot and keep going up, step by step. If each step takes you higher, you stick with it, but if not, you try a different direction. This goes on until you can't climb any higher, reaching a point called a local maximum. In computer terms, Hill Climbing starts with a solution and improves it bit by bit based on how good it is according to a rule. This continues until no more improvements can be made.

Beam Search

You start with a set of possible paths (the beam) and explore them simultaneously. At each step, you keep only the most promising paths based on certain criteria. This process continues until you find a satisfactory solution or exhaust your options.

Both Hill Climbing and Beam Search have their own advantages and disadvantages, and which one you choose depends on what you need for the particular problem you're trying to solve.

VI. Conclusion

In the Marble Solitaire case, we saw how using various search algorithms like priority queue-based search, Best-First Search, and A* showed how different heuristic functions affect their performance. Similarly, when dealing with Uniform Random k-SAT problems, we compared different heuristic functions and search strategies. By comparing the performance of Hill-Climbing, Beam-Search, and Variable-Neighborhood-Descent algorithms, we learned that each has its own strengths and weaknesses depending on the problem at hand. Overall, this study not only enhanced our understanding of heuristic functions and non-traditional search algorithms but also showed how they can be applied practically in the field of AI.

References

[1] https://medium.com/@aksblog/peg-of-solitaire-game-busted-with-ai-c5f73466f8c3
[2] https://www.stat.berkeley.edu/ mossel/teach/206af06/scribes/sep14.pdf

Lab Assignment 3
Travelling Salesman Problem (TSP)

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Python notebook code for this assignment can be found here.

***Abstract*—This report explores the application of Simulated Annealing to solve the Travelling Salesman Problem (TSP) for planning a cost-effective tour of Rajasthan. The TSP involves finding the shortest possible cycle that visits each city exactly once. We aim to optimize the tour route based on the distances between tourist locations in Rajasthan.**

I. Problem Statement

Traveling Salesman Problem (TSP) is a hard problem and is simple to state. Given a graph in which the nodes are locations of cities, and edges are labeled with the cost of traveling between cities, find a cycle containing each city exactly once, such that the total cost of the tour is as low as possible.

For the state of Rajasthan, find out at least twenty important tourist locations. Suppose your relatives are about to visit you next week. Use Simulated Annealing to plan a cost-effective tour of Rajasthan. It is reasonable to assume that the cost of traveling between two locations is proportional to the distance between them.

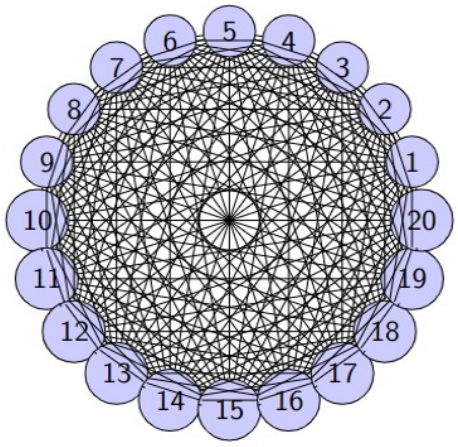
II. Introduction

The Travelling Salesman Problem (TSP) poses a significant challenge in optimization, requiring the determination of the most efficient route that traverses all specified cities exactly once and returns to the starting point. In this report, we employ Simulated Annealing, a stochastic optimization technique inspired by the annealing process in metallurgy, to tackle the complexities of planning a cost-effective tour of Rajasthan for visiting relatives. The primary objective is to minimize the total cost incurred during the tour while ensuring that each tourist destination is included in the itinerary precisely once. By leveraging the power of Simulated Annealing, we aim to efficiently explore the vast search space of possible tour routes, ultimately arriving at an optimal solution that balances cost-effectiveness with comprehensive coverage of Rajasthan's key attractions.

III. Methodology

Simulated Annealing stands as a powerful optimization method that draws inspiration from the annealing process in metallurgy. This method is adept at navigating complex problem spaces by emulating the gradual cooling of metal, eventually reaching a stable state with minimized energy. The algorithm begins with an initial solution and systematically explores neighboring solutions, assessing their suitability based on a probability-driven acceptance criterion. Through iterations, Simulated Annealing dynamically adjusts its exploration strategy, gradually diminishing the likelihood of accepting suboptimal solutions as it approaches an optimal solution. This iterative refinement process empowers Simulated Annealing to efficiently traverse intricate problem landscapes, allowing it to uncover high-quality solutions even in challenging optimization scenarios. By mimicking the annealing process, this approach provides a powerful means of solving complex optimization problems, making it a valuable tool in various fields, including logistics, manufacturing, and scheduling.

In our endeavor to apply the Traveling Salesman Problem (TSP) for planning a comprehensive tour of Rajasthan, meticulous attention was given to data collection. We diligently gathered the latitude and longitude coordinates of twenty key cities within the vibrant state of Rajasthan. These cities, each boasting their own unique charm and historical significance, were selected to encapsulate the rich tapestry of cultural, architectural, and natural wonders that Rajasthan has to offer. From the bustling streets of Jaipur to the tranquil lakes of Udaipur, and from the majestic forts of Jodhpur to the serene landscapes of Mount Abu, each city was carefully curated to ensure a diverse and immersive touring experience. By meticulously documenting the geographic coordinates of these cities, we



Total Possible Paths - **60822550204416000**

Fig. 1. Total Possible Paths

laid the foundation for applying optimization algorithms such as Simulated Annealing to craft an optimal tour itinerary that seamlessly navigates the captivating landscape of Rajasthan, providing travelers with an unforgettable journey through the heart of India's royal heritage.

DataSet: Rajasthan Tourist Places

- **NAME:** Rajasthan
- **COMMENT:** List of 20 tourist places in Rajasthan
- **TYPE:** TSP
- **DIMENSION:** 20
- **EDGE_WEIGHT_TYPE:** EUC_2D

Name	Latitude	Longitude
Jaipur	26.9124	75.7873
Udaipur	24.5854	73.7125
Jodhpur	26.2389	73.0243
Jaisalmer	26.9157	70.9083
Ajmer	26.4499	74.6399
Mount Abu	24.5926	72.7156
Bikaner	28.0176	73.3150
Bharatpur	27.2152	77.5030
Chittorgarh	24.8829	74.6230
Bundi	25.4326	75.6483
Kota	25.2138	75.8648
Shekhawati	27.6195	75.1504
Pali	25.7781	73.3311
Sariska Tiger Reserve	27.3104	76.4389
Ranthambore National Park	26.0173	76.5026
Shakambari Jheel	26.9261	75.0962
Neemrana	27.9797	76.3962
Bhangarh Fort	27.0964	76.2862
Ranthambore Fort	26.0185	76.4557
Bhilwara	26.7748	72.3304

IV. Results

We implemented the Simulated Annealing algorithm to optimize the tour route for visiting twenty tourist locations in Rajasthan. The algorithm was run for a specified number of iterations, with parameters tuned to balance exploration and exploitation. The resulting tour route was evaluated for its total cost, which represents the overall distance traveled.

V. Discussion

The cost-effective tour route produced by Simulated Annealing presents a pragmatic approach to optimizing travel itineraries in Rajasthan. Through the minimization of total distance traveled, our relatives can explore essential tourist destinations while maintaining manageable travel expenses. However, it's essential to acknowledge that the efficacy of the solution may fluctuate based on factors like traffic conditions and time constraints. Despite this variability, Simulated Annealing serves as a valuable tool for generating travel plans that strike a balance between cost-effectiveness and comprehensive exploration of Rajasthan's attractions.

VI. Conclusion

Simulated Annealing proves to be a valuable asset for crafting cost-effective tours in Rajasthan. Through the utilization of this randomized search algorithm, we can adeptly streamline travel routes and curtail expenses for our visiting relatives. Moving forward, there is potential for enhancing the algorithm's performance by fine-tuning its parameters and integrating real-time data sources. These advancements hold promise for refining tour planning accuracy and ensuring an optimal balance between cost efficiency and comprehensive exploration of Rajasthan's diverse attractions.

References

1) Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598), 671-680.
2) Aarts, E. H., & Korst, J. (1989). Simulated annealing and Boltzmann machines: A stochastic approach to combinatorial optimization and neural computing. *Wiley*.

Experiment 4
Game Playing Agent- Minimax- Alpha-Beta Pruning

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Google collab notebook for this assignment can be found here.

***Abstract*—This research paper aims to examine how computers play games using smart strategies. We study how strategies like Minimax and alpha-beta pruning improve computer gameplay in games like Tic-Tac-Toe and Nim. These strategies help computers make smarter decisions by cutting down unnecessary choices. This report helps us understand how computers can play games really well.**

I. Learning Objective

The aim of this experiment is to see how using Minimax and alpha-beta pruning makes computer gameplay better. We want to find out if these methods help computers play games smarter and faster, especially in situations where they have to compete against an opponent.

II. Introduction

Computers and games have always fascinated us, especially when computers can play games just like humans. We introduce some key concepts of game playing and decision-making algorithms.

In this experiment, we cover a few important ideas. First is the Playing Agent, which can be either a person or a computer in a game who makes choices. Then, we talk about two key strategies: Minimax and Alpha-Beta Pruning. Minimax is like a smart decision-making process for games, especially to avoid losing badly. Alpha-Beta Pruning is a trick to make Minimax work faster by skipping some unnecessary steps. These concepts are the building blocks for making computer gameplay better.

III. Problem I: Noughts and Crosses (Tic-Tac-Toe)

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

Tic-Tac-Toe is a two-player game played on a 3x3 grid, where players take turns marking spaces with their respective symbols, usually X and O. The objective is to form a line of three of your symbols horizontally, vertically, or diagonally. The game keeps going until one player gets three in a row and wins, or until the grid is full with no winner, resulting in a draw. If both players make the best moves they can, the game will always end in a draw.

To calculate the size of the game tree for Noughts and Crosses, we need to consider two main factors: branching factor and depth.

- **Branching Factor:** The branching factor represents the number of available moves for each player at any given point. In a 3x3 grid, there are initially 9 empty cells, so the branching factor is 9 for the first move. As the game progresses, the number of empty cells decreases, leading to a decreasing branching factor.
- **Depth:** Depth refers to how many moves are made in a game until it ends. In the worst-case scenario, where the game reaches a draw, each player will make up to 9 moves, completely filling the grid. Thus, the maximum depth of the game tree can reach 9 levels.

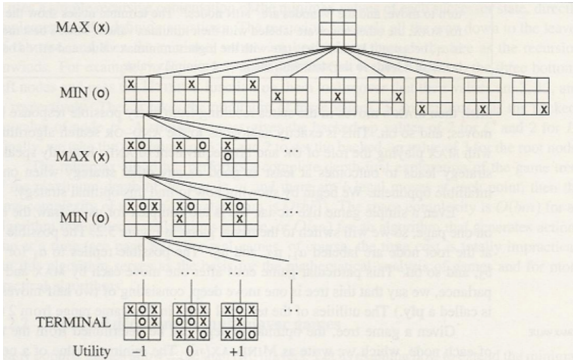


Fig. 1. Game tree for noughts and crosses

S_0 represents the initial state of a game of Tic-Tac-Toe.

$$S_0 = \begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

Each level represents a player's turn. Nodes depict the game state after a player's move, starting from the initial empty grid at the root node. Branches represent possible moves, expanding exponentially with each level. Leaf nodes signify terminal states, such as win, lose, or draw.

The game tree initially has 9! (362,880) edges, which includes games that extend beyond a victory. Removing these games leaves 255,168 edges. This number might appear unexpectedly large for a simple game like Noughts and Crosses due to several reasons.

1) There's only one starting position.
2) Each player has 9 possible moves in the first turn and 8 in subsequent turns because one square is already occupied.
3) Players taking turns to make moves effectively doubles the number of possible positions at each level of the game tree.

When we combine similar game boards, we end up with 26,830 edges, mostly because we're dealing with rotations and reflections. But this count is too high because it includes some illegal positions and doesn't consider symmetrical scenarios, like flipping the board. In reality, there are fewer unique positions than this count suggests.

IV. Problem II: The Nim Game

Nim is a two-player game involving heaps of objects, such as stones, coins, or matchsticks. Players take turns removing objects from one or more heaps according to set rules. Typically, a player can take any number of objects from a single heap during their turn. The game continues until no objects remain. The player who takes the last object wins.

Algorithm

1) Define Initial State: Begin with the initial configuration of the game, specifying the number of objects in each pile.
2) Build Game Tree: Construct a game tree showing every possible move and counter-move from the game's start. Each point on the tree represents a game situation, and each line between points shows a possible move from one situation to another.
3) Assign Terminal Values: Assign terminal values to the leaf nodes of the tree based on whether the state is a win, loss, or draw for player-2. In Nim, a state where player-2 has no legal move left signifies a win, while a state where player-1 has no legal move left results in a loss.

Fig. 2. Nim game initial State

Minimax

1) Start at the root node of the game tree.
2) For each node:
   - Assign the terminal value to the node if it is a leaf node.
   - If the node represents a MAX node (indicating player 1's turn), select the child node with the highest value, which represents player 1's best move.
   - If the node represents a MIN node (indicating player 2's turn), choose the child node with the minimum value, reflecting player 2's optimal move.
   - Update the values up the tree until reaching the root node.

Decision making: Evaluate the computed values to determine the best move for player 1. Since player 2 always seeks to maximize their chances of winning, player 1's optimal move is determined by selecting the move that minimizes the maximum potential loss for player 2.

Game Outcome: Regardless of player 1's move, player 2 responds optimally to uphold their winning strategy. This ensures that player 2 will always have a winning response, resulting in a victory for player 2.

This algorithm shows how the MINIMAX method can be used to analyze Nim, revealing that player 2 will always win, no matter what player 1 does. It's because player 2 can always make the best move to stay ahead.

V. Problem III: Alpha beta pruning

Alpha-beta pruning is an optimization technique used in game trees to reduce the number of nodes that need to be evaluated. It's based on the Minimax algorithm, which aims to find the best move for a player assuming the opponent also plays optimally.

- **Alpha:** The alpha value represents the best value that the maximizing player (e.g., player 1) can achieve so far

along the path of any of the possible moves from this state.

- **Beta:** The beta value represents the best value that the minimizing player (e.g., player 2) can achieve so far along the path of any of the possible moves from this state.

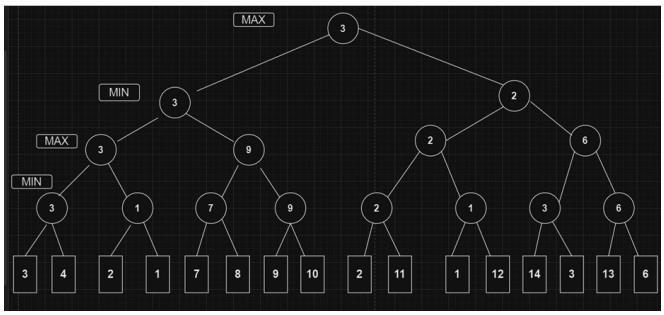


Fig. 3. Minimax tree

The code we've implemented uses a Tic-Tac-Toe board where rows and columns start counting from 0. So, in this setup, the top-left corner of the board is represented by the coordinates (0, 0). As a result, the algorithm suggests placing an X in the top-left corner as the best move after analyzing the game situation.

It uses the MINIMAX algorithm with alpha-beta pruning to find the best move for player X in Tic-Tac-Toe. It looks at all possible future game scenarios up to a certain depth, ensuring the best decisions are made.

VI. Problem IV: Time Complexity

When using alpha-beta pruning with perfectly ordered leaf nodes, the algorithm visits nodes in a depth-first manner, exploring branches of the game tree until it reaches a terminal state or a cutoff condition. In the best-case scenario, where leaf nodes are perfectly ordered, the algorithm can prune entire subtrees without fully exploring them, significantly reducing the time complexity.

Let's define the recurrence relation:

N(d): Number of nodes visited at depth

b: Effective branching factor (average number of children per node).

m: Depth of the tree

To simplify, let's assume that the effective branching factor b remains constant at each level of the tree.

Recurrence relation: At each level d, the algorithm explores at most b children (assuming perfect ordering). However, due to pruning, the algorithm may explore fewer nodes depending on the values of alpha and beta.
Therefore,

$$N(d) = b \cdot N(d-1)$$

The number of nodes explored at depth d is directly related to the number of nodes explored at depth $d-1$, multiplied by the effective branching factor b.
Solving the relation we get,

$$N(d) = b^d$$

The total visited nodes throughout the search is the sum of nodes explored at each depth from 1 to m.

$$\sum_{d=1}^{m} N(d) = \sum_{d=1}^{m} b^d = \sum_{d=1}^{m} \frac{b^{m+1} - b}{b-1}$$

Therefore, the time complexity of alpha-beta pruning under perfect ordering of leaf nodes is O(b^m)

Considering the depth m halved due to alpha-beta pruning, the time complexity becomes $O(b^{m/2})$, where b is the branching factor.

VII. Conclusion

We closely studied the Noughts and Crosses game, understanding its structure, rules, and the different tactics players can use to outsmart their opponents and secure victory. We also explained the game of Nim, showing that no matter what Player-1 does, Player-2 has a better chance of winning by thinking carefully about the game and making the best moves possible. We implemented two fundamental algorithms, MINIMAX and alpha-beta pruning, to automate decision-making processes in Noughts and Crosses. Alpha-beta pruning helps make calculations faster by cutting down on unnecessary work, especially when game states are ordered optimally.

References

[1] https://en.wikipedia.org/wiki/Game$_complexity$
[2] https://www.cs.swarthmore.edu/ mitchell/classes/cs63/s24/labs/03.html
[3] https://medium.com/chat-gpt-now-writes-all-my-articles/advanced-ai-alpha-beta-pruning-in-python-through-tic-tac-toe-70bb0b15db05

Experiment 5
Graphical models for inference under uncertainity and Naive Bayes classification

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Google collab notebook for this assignment can be found here.

***Abstract*—This report explores how we can use graphs to understand situations where things aren't certain. It discusses creating models called Bayesian Networks using R, a programming language. This report explores how we can use these ideas practically, using the R programming language and a package called bnlearn. The goal is to learn the structure and CPTs of a Bayesian Network from a dataset of student grades, and subsequently use this network for inference and classification tasks.**

I. Learning Objective

Understand the graphical models for inference under uncertainty, build Bayesian Network in R, Learn the structure and CPTs from Data, naive Bayes classification with dependency between features.

II. Introduction

Graphical models are great for handling uncertainty because they offer a structured way to represent and work with complicated probability information. Among them, Bayesian Networks are particularly good at showing how things relate to each other. It's like a flowchart where each point (or node) is a random variable, and each arrow (or edge) shows how one variable affects another. Each node has a probability function that tells us the likelihood of different outcomes. Bayesian networks help us see how different things are connected.

III. Probelm I: Dependencies Between Courses

Problem Statement: Consider grades earned in each of the courses as random variables and learn the dependencies between courses.

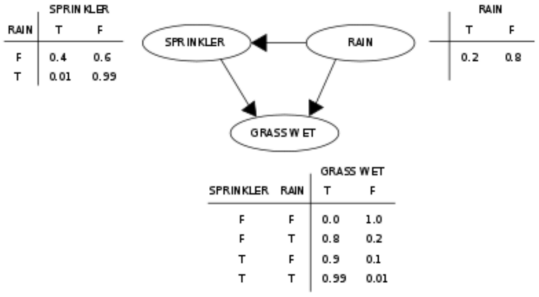


Fig. 1. Example of a Bayesian network

A. Bayesian Network

Bayesian networks are a way of modeling relationships between different events or variables through a directed acyclic graph (DAG).

Key Components:

- Nodes: Random variable
- Edges: Conditional dependencies among variables
- Conditional Probability Tables (CPTs): Associated with each node is a conditional probability table, which specifies the probability distribution of a node given its parents in the graph. It quantifies how the variables are related.

$$\text{Cause} \rightarrow \text{Effect}$$

They are particularly useful for modeling and reasoning under uncertainty. These networks can be employed for tasks such as classification, prediction, diagnosis, and decision-making by utilizing probabilistic inferences.

Naive Bayes: Naive Bayes is a machine learning algorithm used for classification tasks. The **Naive** part comes from a simplifying assumption the algorithm makes: it

	EC100	EC160	IT101	IT161	MA101	PH100	PH160	HS101	QP
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	BC	CC	BB	BC	CC	BC	AA	BB	y
2	CC	BC	BB	BB	CC	BC	AB	BB	y
3	AB	BB	AB	AB	BB	CC	BC	AB	y
4	BC	CC	BB	BB	BB	BB	BC	BB	y
5	BC	AB	CD	BC	BC	BC	BC	CD	y
6	DD	CC	DD	CD	CD	CC	BC	BC	n

Fig. 2. Dependency between courses

assumes that all the features are independent of each other.

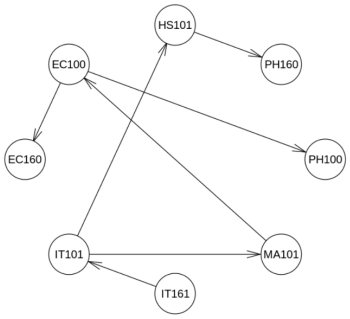


Fig. 3. Dependency Graph of different courses

The Bayesian network was created using the Hill-Climbing algorithm with Cooper and Herskovits' K2 scoring metric. It has 8 nodes and 7 arrows pointing in different directions, with an average Markov blanket size of 1.75 and an average branching factor of 0.88. The learning process was improved and tested 105 times. The network structure shows how variables like IT161, IT101, MA101, HS101, EC100, PH160, EC160, and PH100 are connected and affect each other's probabilities.

IV. Problem II: Learning the CPTs for Each Course Node

Problem Statement: Using the data, learn the CPTs for each course node.

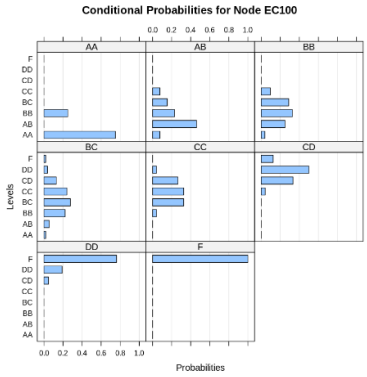


Fig. 4. Conditional probabilities for Node EC100

```
  Bayesian network parameters

  Parameters of node EC100 (multinomial distribution)

Conditional probability table:

    MA101
EC100         AA          AB          BB          BC          CC          CD
   AA 0.75000000 0.07692308 0.03846154 0.01851852 0.00000000 0.00000000
   AB 0.00000000 0.46153846 0.25000000 0.05555556 0.00000000 0.00000000
   BB 0.25000000 0.23076923 0.32692308 0.22222222 0.04081633 0.00000000
   BC 0.00000000 0.15384615 0.28846154 0.27777778 0.32653061 0.00000000
   CC 0.00000000 0.07692308 0.09615385 0.24074074 0.32653061 0.04166667
   CD 0.00000000 0.00000000 0.00000000 0.12962963 0.26530612 0.33333333
   DD 0.00000000 0.00000000 0.00000000 0.03703704 0.04081633 0.50000000
   F  0.00000000 0.00000000 0.00000000 0.01851852 0.00000000 0.12500000
    MA101
EC100         DD          F
   AA 0.00000000 0.00000000
   AB 0.00000000 0.00000000
   BB 0.00000000 0.00000000
   BC 0.00000000 0.00000000
   CC 0.00000000 0.00000000
   CD 0.04761905 0.00000000
...
   CD 0.13043478 0.08333333
   DD 0.52173913 0.58333333
   F  0.04347826 0.33333333
```

```
  Parameters of node EC160 (multinomial distribution)

Conditional probability table:

    EC100
EC160         AA          AB          BB          BC          CC          CD
   AA 0.42857143 0.22727273 0.05714286 0.04166667 0.00000000 0.00000000
   AB 0.42857143 0.22727273 0.08571429 0.04166667 0.08333333 0.00000000
   BB 0.14285714 0.31818182 0.20000000 0.22916667 0.08333333 0.03448276
   BC 0.00000000 0.22727273 0.42857143 0.43750000 0.36111111 0.17241379
   CC 0.00000000 0.00000000 0.22857143 0.25000000 0.30555556 0.34482759
   CD 0.00000000 0.00000000 0.00000000 0.00000000 0.11111111 0.27586207
   DD 0.00000000 0.00000000 0.00000000 0.00000000 0.05555556 0.17241379
   F  0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
    EC100
EC160         DD          F
   AA 0.00000000 0.00000000
   AB 0.00000000 0.00000000
   BB 0.05000000 0.00000000
   BC 0.00000000 0.00000000
   CC 0.25000000 0.02857143
   CD 0.55000000 0.40000000
   DD 0.15000000 0.34285714
   F  0.00000000 0.22857143
```

Fig. 5. Conditional Probability Tables

Similarly, we calculated Conditional Probability Tables (CPTs) for every course in the dataset. These tables show the likelihood of different outcomes for each course, based on the outcomes of its related courses in the Bayesian network. This helps us understand how the courses influence each other's outcomes, making it easier to make informed decisions based on the data.

V. Problem III: Predicting Student Grades

Problem Statement: What grade will a student get in PH100 if he earns DD in EC100, CC in IT101 and CD in MA101.

With the Bayesian Network and its Conditional Probability Tables (CPTs) that we created, we can predict a student's grade in PH100. We do this by putting in the grades we already know and then figuring out the chances of getting different grades in PH100 based on that information.
The implemented code predicts the grade in PH100 based on the given evidence.

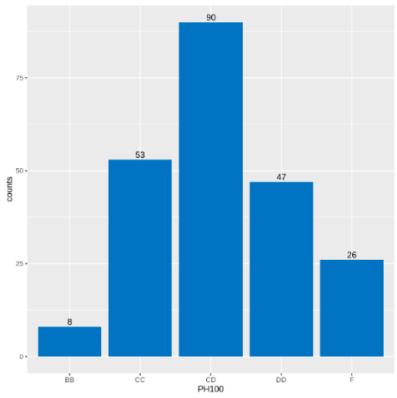


Fig. 6. Predicted Grade in PH100

VI. Probelm IV: Building a Naive Bayes Classifier

Problem Statement: The last column in the data file indicates whether a student qualifies for an internship program or not. From the given data, take 70 percent data for training and build a naive Bayes classifier (considering that the grades earned in different courses are independent of each other) which takes in the student's performance and returns the qualification status with a probability. Test your classifier on the remaining 30 percent data. Repeat this experiment for 20 random selection of training and testing data. Report results about the accuracy of your classifier.

A Naive Bayes classifier is created using 70% of the data for training. The classifier assumes that the grades earned in different courses are independent of each other. Then, the classifier is tested on the remaining 30% of the data. This process is repeated 20 times with randomly chosen training and testing data, and the accuracy of the classifier is recorded.

Achieved average accuracy = 96.93%

```
Accuracy for iteration 1 : 97.14 %
Accuracy for iteration 2 : 98.57 %
Accuracy for iteration 3 : 97.14 %
Accuracy for iteration 4 : 100 %
Accuracy for iteration 5 : 98.57 %
Accuracy for iteration 6 : 98.57 %
Accuracy for iteration 7 : 98.57 %
Accuracy for iteration 8 : 98.57 %
Accuracy for iteration 9 : 94.29 %
Accuracy for iteration 10 : 98.57 %
Accuracy for iteration 11 : 97.14 %
Accuracy for iteration 12 : 91.43 %
Accuracy for iteration 13 : 95.71 %
Accuracy for iteration 14 : 97.14 %
Accuracy for iteration 15 : 95.71 %
Accuracy for iteration 16 : 97.14 %
Accuracy for iteration 17 : 95.71 %
Accuracy for iteration 18 : 97.14 %
Accuracy for iteration 19 : 94.29 %
Accuracy for iteration 20 : 97.14 %
```

Fig. 7. Accuracy achieved in each iteration

VII. Problem V: Considering Dependencies in Naive Bayes Classification

Problem Statement: Repeat 4, considering that the grades earned in different courses may be dependent.

In this case, we adjust the Naive Bayes classifier to recognize that there could be connections between the grades in different courses. This means we change how the classifier calculates probabilities to consider these potential relationships.

Achieved average accuracy = 97.61%

```
Accuracy for iteration 1 : 97.14 %
Accuracy for iteration 2 : 98.57 %
Accuracy for iteration 3 : 97.14 %
Accuracy for iteration 4 : 100 %
Accuracy for iteration 5 : 98.57 %
Accuracy for iteration 6 : 98.57 %
Accuracy for iteration 7 : 98.57 %
Accuracy for iteration 8 : 98.57 %
Accuracy for iteration 9 : 94.29 %
Accuracy for iteration 10 : 98.57 %
Accuracy for iteration 11 : 97.14 %
Accuracy for iteration 12 : 91.43 %
Accuracy for iteration 13 : 95.71 %
Accuracy for iteration 14 : 97.14 %
Accuracy for iteration 15 : 95.71 %
Accuracy for iteration 16 : 97.14 %
Accuracy for iteration 17 : 95.71 %
Accuracy for iteration 18 : 97.14 %
Accuracy for iteration 19 : 94.29 %
Accuracy for iteration 20 : 97.14 %
```

Fig. 8. Accuracy achieved in each iteration(considering dependencies)

VIII. Conclusion

Studying graphical models, Bayesian Networks, and Naive Bayes classification helps us predict outcomes in uncertain situations. Understanding how things are connected and estimating these connections accurately in a Bayesian Network is crucial for reliable predictions. Also, deciding whether to assume independence or consider relationships between features makes a big difference in how well a Naive Bayes classifier works.

References

[1] http://gauss.inf.um.es/umur/xjurponencias/talleres/J3.pdf
[2] https://www.bnlearn.com/

Lab Assignment 6
Implementing Expectation Maximization for Learning Parameters of a Hidden Markov Model

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Python notebook code for this assignment can be found here.

***Abstract*—This report explores the implementation of the Expectation Maximization (EM) routine for learning parameters of a Hidden Markov Model (HMM). The learning objectives include understanding the EM framework and its application to problems with hidden or partial information. We detail the problem statement, which involves implementing routines for learning HMM parameters and performing experiments based on a provided reference. Additionally, we conduct experiments involving ten bent coins to determine unknown bias values and perform clustering using EM on a dataset with real values. Through rigorous experimentation and analysis, we aim to gain insights into the effectiveness and versatility of the EM framework in handling problems with hidden information.**

I. Problem Statement

- Read through the reference carefully. Implement routines for learning the parameters of HMM given in section 7. In section 8, “A not-so-simple example”, an interesting exercise is carried out. Perform a similar experiment on “War and Peace” by Leo Tolstoy.

- Ten bent (biased) coins are placed in a box with unknown bias values. A coin is randomly picked from the box and tossed 100 times. A file containing results of five hundred such instances is presented in tabular form with 1 indicating head and 0 indicating tail. Find out the unknown bias values. (2020-ten-bent-coins.csv) To help you, a sample code for two bent coin problems along with data is made available in the work folder: two-bent-coins.csv and embentcoinsol.m

- A point set with real values is given in 2020-em-clustering.csv. Considering that there are two clusters, use EM to group together points belonging to the same cluster. Try and argue that k-means is an EM algorithm.

II. Introduction

In this report, we explore the implementation of the Expectation Maximization (EM) routine for learning parameters of a Hidden Markov Model (HMM). The learning objectives include understanding the EM framework and its application to problems with hidden or partial information. We begin by discussing the problem statement and the tasks to be performed based on the reference provided. We then delve into the implementation of routines for learning the parameters of the HMM, as outlined in section 7 of the reference. Subsequently, we undertake an interesting experiment based on ”War and Peace” by Leo Tolstoy, following a similar approach outlined in section 8 of the reference.

III. Implementation of EM for HMM

A Markov process is a random process that is indexed by time. It is a stochastic model that describes a sequence of possible events. The probability of each event depends only on the state attained in the previous event.

Hidden Markov Models (HMMs) are probabilistic models used for modeling sequential data, where the underlying system is assumed to be a Markov process with unobservable (hidden) states.

We detail the implementation of the EM routine for learning the parameters of the HMM. This involves iterative steps of expectation and maximization to update the parameters until convergence is achieved.

IV. Experiment on "War and Peace"

Following the experiment outlined in section 8 of the reference, we conducted a similar analysis of "War and Peace" by Leo Tolstoy. We aim to extract hidden information using the EM framework and derive insights from the text.

V. Experiment with Ten Bent Coins

Expectation Maximization (EM) is an iterative algorithm used to find maximum likelihood or maximum posterior estimates of parameters in statistical models where some variables are unobserved or missing. It is particularly useful in cases where the model depends on latent variables, which are variables that are not directly observed but are inferred from observed data.

We analyze the dataset containing the results of coin tosses for ten bent coins and determine the unknown bias values using EM. We discuss the methodology, implementation details, and the results obtained.

A. How the EM algorithm works

- **Expectation Step (E-step):** In this step, you make an educated guess about the missing information based on the data you have. You calculate the probabilities or likelihoods of different scenarios given what you know.
- **Maximization Step (M-step):** Now that you have your educated guess from the E-step, you use it to update your knowledge about the missing information. You adjust your parameters or guesses to better fit the data.
- **Iteration:** Steps 2 and 3 are repeated iteratively until the algorithm converges, meaning that the parameter estimates stop changing significantly between iterations or until a maximum number of iterations is reached.

Experiment 1: We see which coin is flipped.

coin	flips	# coin 1 heads	# coin 2 heads
B	HTTTHHTHTH	0	5
A	HHHHTHHHHH	9	0
A	HTHHHHHTHH	8	0
B	HTHTTTHHTT	0	4
A	THHHTHHHTH	7	0

Coin 1	Coin 2
	5H,5T
9H,1T	
8H,2T	
	4H,6T
7H,3T	
24H,6T	9H,11T

This means that if we toss coin A 80% it will come up with Heads.

Experiment 2: We don't see which coin is flipped

coin	flips	# coin 1 heads	# coin 2 heads
?	HTTTHHTHTH	?	?
?	HHHHTHHHHH	?	?
?	HTHHHHHTHH	?	?
?	HTHTTTHHTT	?	?
?	THHHTHHHTH	?	?

flips	probability it was coin C1	probability it was coin C2	# heads attributed to C1	# heads attributed to C2
HTTTHHTHTH	0.45	0.55	5 * 0.45 = 2.25	5 * 0.55 = 2.75
HHHHTHHHHH	0.8	0.2	9 * 0.8 = 7.2	9 * 0.2 = 1.8
HTHHHHHTHH	0.73	0.27	8* 0.73 = 5.84	2 * 0.27 = 2.16
HTHTTTHHTT	0.35	0.65	4 * 0.35 = 1.4	4 * 0.65 = 2.6
THHHTHHHTH	0.65	0.35	7 * 0.65 = 4.55	7 * 0.35 = 2.45

VI. Clustering using EM

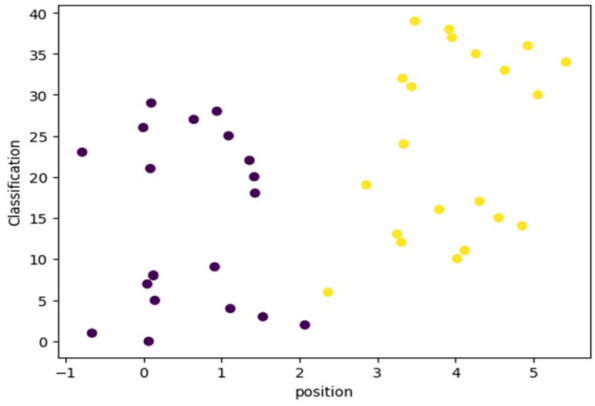
K-means is another popular algorithm used for clustering data points into groups or clusters. It's a simpler and more

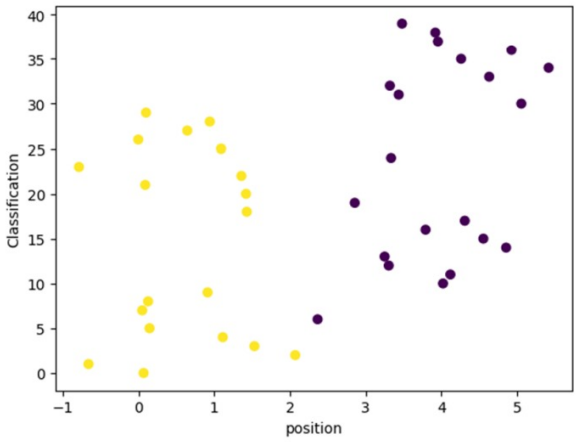
computationally efficient method compared to the EM algorithm, particularly when dealing with large datasets.

A. How K-means works:

- **Initialization:** We choose K initial cluster centroids randomly from the data points. K represents the number of clusters we want to identify.
- **Assignment Step:** We assign each data point to the nearest centroid. This is typically done by calculating the Euclidean distance (or other distance metrics) between each data point and each centroid and assigning the data point to the nearest centroid.
- **Update Step:** Recalculate the centroids of the clusters by taking the mean of all data points assigned to each centroid. This moves the centroids to the center of their respective clusters.
- **Iteration:** Repeating the assignment and update steps iteratively until convergence criteria are met. Typically, convergence occurs when the centroids no longer change significantly between iterations or when a maximum number of iterations is reached.

Using the provided dataset (2020-em-clustering.csv), we perform clustering with EM to group together points belonging to the same cluster. We compare the results with k-means clustering and discuss the relationship between the two algorithms.





VII. Conclusion

In conclusion, we have successfully implemented the EM routine for learning the parameters of an HMM and applied it to various experiments as outlined in the problem statement. Through rigorous experimentation and analysis, we have gained insights into the effectiveness and versatility of the EM framework in handling problems with hidden or partial information. Future work may involve further exploration of advanced EM techniques and their application to real-world datasets.

Lab Assignment 7
Artificial Intelligence

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

***Abstract*—One popular supervised learning technique in machine learning is the decision tree, which models and predicts outcomes based on input data. It resembles a tree. structure in which every leaf node denotes the final judgment or prediction, every branch matches an attribute value, and every internal node tests an attribute. The supervised learning category includes the decision tree method. They can be applied to the resolution of classification and regression issues.**

I. Introduction

To begin, decision trees function akin to flowcharts aiding in decision-making by breaking down data into more manageable segments. Initially, we employ entropy to guide these divisions, which measures the impurity or mixedness of our data. The objective is to minimize entropy with our splits, essentially aiming for decisions that result in purer groups.

To ensure the accuracy of our decision tree, we are implementing cross-validation, akin to double-checking our work. Rather than relying solely on one dataset, we partition our data into multiple segments, training our decision tree on some segments and testing it on the rest. This process helps us assess the performance of our decision tree across different datasets, ensuring it does not merely excel at memorizing one specific dataset.

Additionally, we are exploring an alternative approach using the Gini index instead of entropy. The Gini index also evaluates the impurity of our data but does so in a slightly different manner than entropy. It's akin to trying a different tool to determine if it yields better outcomes.

Lastly, we are addressing a common concern known as overfitting. Overfitting occurs when our decision tree becomes overly adept at fitting the training data, yet fails to generalize well to new, unseen data. To tackle this, we are adjusting the maximum depth of our decision tree. Think of it as trying on different sizes of clothing: if the tree is too deep, it may fit the training data perfectly but struggle with new data, while if it's too shallow, it may not capture enough complexity from the data. Our goal is to find the ideal depth where the tree fits just right.

II. Concepts Used

- **Decision Trees:** Decision Trees act as decision-making flowcharts, directing each decision along a different path. In machine learning, decision trees use data features to make decisions. At each step, the tree evaluates a feature and determines how to split the data into smaller, more homogeneous groups. This process continues until the data is divided into groups that are as pure as possible, containing similar instances of the target variable in classification tasks or similar values in regression tasks.
- **Entropy:** Entropy quantifies the impurity or randomness of a data group at a specific node in the decision tree. Lower entropy signifies more uniformity, indicating that the data at that point is more homogeneous. In contrast, higher entropy suggests greater diversity in the data at that node.
- **Cross-validation:** Cross-validation is a method used to assess machine learning model performance. It entails dividing the data into multiple subsets, using some for training the model and reserving others for testing. By iterating through various training and testing subsets, cross-validation provides a more robust estimate of how well the model will perform on unseen data.
- **Gini Index:** The Gini Index is an alternative impurity measure used in decision trees. It gauges the probability that a randomly selected element from the set would be incorrectly labeled if labeled according to the distribution of labels in the subset. In practical terms, it helps determine how to split data in a decision tree.
- **Overfitting:** Overfitting arises when a machine learning model captures noise or irregularities in the training data instead of learning the underlying patterns that generalize well to new data. In decision trees, overfitting can occur if the tree becomes overly intricate and memorizes the training data rather than learning from its overall structure. It's crucial to monitor for overfitting and take preventative

measures, such as limiting model complexity or using techniques like cross-validation.

III. Objective

A car dataset is uploaded in the lab-work folder with a description specifying all the attributes, the values that each attribute can take, and the class labels.

1) Randomly select 60 percent of labeled data (from each class) for constructing the tree (training). Test for the rest of 40 percent data. Find out the accuracy of the classification tree with the help of a confusion matrix and F-score. Use the entropy measure for the selection of attributes.
2) Repeat the above exercise 20 times. Calculate the average accuracy of classification.
3) Repeat steps 1 and 2 with the Gini index as a measure for the selection of attributes.
4) Repeat steps 1, 2, and 3 considering 70 and 80 percent data (random selection) for training.
5) Describe the problem of overfitting in your words with an example created from the dataset.

IV. Theory

Problem 1 Explanation

Next, categorical variables—which stand for non-numerical data such as labels or categories—are converted into numerical form with a method like one-hot encoding. This is an important stage because the majority of machine learning algorithms need numerical input data in order to compute and provide predictions. Following encoding, the encoded data structure is separated into the features (independent variables) and the target variable (dependent variable, or class labels). The target variable represents the outcome, or the quantity that has to be anticipated, whereas the features represent the raw data utilized to make predictions.

Next, the data is divided into a training set and a testing set. The training set is used to train the machine learning model; it usually consists of a greater fraction of the data (such as 60% or 70%). The testing set, or the remaining fraction of the data, such as 40% or 30%, is what's utilized to assess how well the trained model performs when applied to untested data. To guarantee that the class distributions in the training and testing sets are proportionate to the original dataset, the splitting process is frequently carried out in a stratified fashion. In order to avoid bias and guarantee that the model is trained and assessed using representative data, this is crucial.

Next, a machine learning model with certain hyperparameters or configurations is constructed, like a decision tree classifier. These hyperparameters can be the maximum tree depth, the criterion for node splitting (e.g., entropy or Gini index), or other parameters that govern the complexity and behavior of the model.

The machine learning library or module provides a fitting or training method that is used to train the model on the training data. The model gains the ability to predict new, unseen data during this phase by discovering patterns and correlations between the features and the target variable.

After training, the model is used to make predictions on the testing data using a prediction method. The predicted class labels or values are then compared with the actual class labels or values in the testing set to evaluate the model's performance. Performance metrics, such as the confusion matrix and the F1-score, are calculated to quantify the model's accuracy, precision, recall, and overall classification performance. The confusion matrix provides a detailed breakdown of the model's predictions, showing the counts of true positives, true nega tives, false positives, and false negatives for each class. The F1-score combines precision and recall into a single score, ranging from 0 to 1, with higher values indicating better performance. The results of the performance evaluation are typically printed or displayed for analysis and interpretation

Problem 2 Explanation

Categorical variables in the dataset are then encoded into numerical form using a technique like one-hot encoding. This step is crucial because most machine learning algorithms require numerical input data.

After encoding, the features (independent variables) and the target variable (dependent variable or class labels) are separated from the encoded DataFrame.

The process is repeated multiple times (e.g., 20 iterations) to account for the randomness involved in splitting the data and training the model. In each iteration, the following steps are performed:

1) The data is split into training (e.g., 60%) and testing (e.g., 40%) sets, ensuring that the class distributions in both sets are proportional to the original dataset.
2) A decision tree classifier is created with a specified criterion (e.g., entropy or Gini index) and a range of maximum depth values.
3) The classifier is trained on the training data using the fit method.
4) Predictions are made on the test data using the predict method.
5) The performance of the classifier is evaluated using metrics like the confusion matrix and the F1-score.

After all iterations, the average performance metrics (e.g., F1-score and confusion matrix) are calculated and printed for each maximum depth value tested.

Finally, the maximum depth value that yields the highest average F1-score is identified and reported as the optimal value for the decision tree classifier on the given dataset.

This process helps in finding the optimal hyperparameters (e.g., maximum depth) for the decision tree classifier, ensuring that the model achieves the best possible classification performance on the dataset while mitigating issues like overfitting.

Problem 3 Explanation

After encoding, the features (independent variables) and the target variable (dependent variable or class labels) are separated from the encoded data structure. The features represent the input data used for making predictions, while the target variable represents the output or the quantity that needs to be predicted.

The data is then split into two subsets: a training set and a testing set. The training set, which typically comprises a larger portion of the data (e.g., 60% or 70%), is used to train the machine learning model. The testing set, which is the remaining portion of the data (e.g., 40% or 30%), is used to evaluate the performance of the trained model on unseen data.

The splitting process is often done in a stratified manner, ensuring that the class distributions in both the training and testing sets are proportional to the original dataset. This is important to prevent bias and ensure that the model is trained and evaluated on representative data.

A machine learning model, such as a decision tree classifier, is then created with specific hyperparameters or configurations. These hyperparameters can include the criterion for splitting nodes (e.g., entropy or Gini index), the maximum depth of the tree, or other parameters that control the model's complexity and behavior.

The model is trained on the training data using a fitting or training method provided by the machine learning library or module. During this process, the model learns patterns and relationships between the features and the target variable, enabling it to make predictions on new, unseen data.

After training, the model is used to make predictions on the testing data using a prediction method. The predicted class labels or values are then compared with the actual class labels or values in the testing set to evaluate the model's performance.

Performance metrics, such as the confusion matrix and the F1-score, are calculated to quantify the model's accuracy, precision, recall, and overall classification performance. The confusion matrix provides a detailed breakdown of the model's predictions, showing the counts of true positives, true negatives, false positives, and false negatives for each class. The F1-score combines precision and recall into a single score, ranging from 0 to 1, with higher values indicating better performance.

The process can be repeated multiple times to account for the randomness involved in splitting the data and training the model. In each iteration, the steps mentioned above are performed, and the performance metrics are stored. After all iterations, the average performance metrics are calculated and reported.

This process helps in assessing the overall performance of the machine learning model on the dataset by running multiple iterations and averaging the results. It accounts for the variability introduced by the random splitting of data and provides a more reliable estimate of the model's performance.

Here is the Google Colab Link for the Experiment:
Click_Here

REFERENCES

[1] Denoise Benchmark. Retrieved from http://www.cs.utoronto.ca/~strider/Denoise/Benchmark/

[2] D. J. Fleet and M. J. Black, "Image denoising via random walk," in *Proceedings of the British Machine Vision Conference*, 2006, pp. 701-710. Retrieved from http://www.cs.toronto.edu/~fleet/research/Papers/BMVC_denoise.pdf

[3] O. F. Sener, "MRF Image Denoising," 2012. Retrieved from https://web.cs.hacettepe.edu.tr/~erkut/bil717.s12/w11a-mrf.pdf

[4] D. J. C. MacKay, "Information Theory, Inference and Learning Algorithms," 2003. Retrieved from http://www.inference.phy.cam.ac.uk/mackay/itila/

Lab Assignment 8
Artificial Intelligence

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

I. Learning objective

Basics of data structure needed for state-space search tasks and use of random numbers required for MDP and RL.

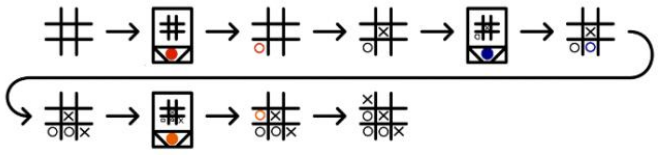
II. Problem Statement

Read the reference on MENACE by Michie and check for its implementations. Pick the one that you like the most and go through the code carefully. Highlight the parts that you feel are crucial. If possible, try to code the MENACE in any programming language of your liking.

III. Introduction

A. What is MENACE

MENACE, which stands for Matchbox Educable Noughts And Crosses Engine, was a 1960s analog computer created by Donald Michie using 304 matchboxes. It was intended to play Tic-Tac-Toe against human opponents by returning a move for any given state of play. Through reinforcement learning, it would gradually ”learn” to improve its approach.Menace learns to play noughts and crosses by using 304 matchboxes, each packed with colored beads. Michie wanted to show that a computer could ”learn” from mistakes and successes in order to improve at a particular task.



Menace is made up of 304 matchboxes, which stand for every possible configuration of a noughts and crosses board that could be encountered during play. These matchboxes are filled with a variety of colored beads, each of which represents a move Menace could make on the matching board layout.

B. How MENACE works?

Menace ”learns” how to play noughts and crosses by playing against another player frequently. Each time, it improves its approach to the point that, after a certain amount of games, it is nearly flawless and its opponent can only draw or lose against it. In a manner similar to how children learn, the learning process entails being ”punished” for losing and ”rewarded” for drawing or winning.

In order to minimize the quantity of matchboxes needed for construction, MENACE always goes first. On a matchbox, every potential game position that MENACE might encounter is depicted. There are several different colored beads in every box. Every color denotes a potential move that MENACE might make from that position.

Finding the box containing the current board position is necessary before utilizing MENACE to perform a move. After that, the operator opens the package by shaking it. MENACE plays at the bead’s color-corresponding position at the front of the box.

C. How MENACE learns?

Consider that the picked beads are taken out of the boxes in the event that MENACE loses the game. MENACE has learnt from this and is therefore less likely to choose the same colors again. Had MENACE prevailed, each box would have received three more beads in the selected color, motivating MENACE to repeat the process. Every box gets one extra bead if the game is a draw.

Four beads of each color are placed in the first move box, three in the third, two in the fifth, and one in the last move box when MENACE initially starts. Later moves are discouraged even more when one bead is removed from each box upon losing. Since the later moves are more likely to have resulted in the loss, this aids MENACE in learning faster.

Following a few games, it’s possible that some of the boxes become empty. MENACE steps down if one of these boxes is

to be checked. The first move box may run out of beads when you play against more experienced players. To give MENACE more time to learn before it begins to resign in this instance, additional beads should be added to the earlier boxes when it is reset.

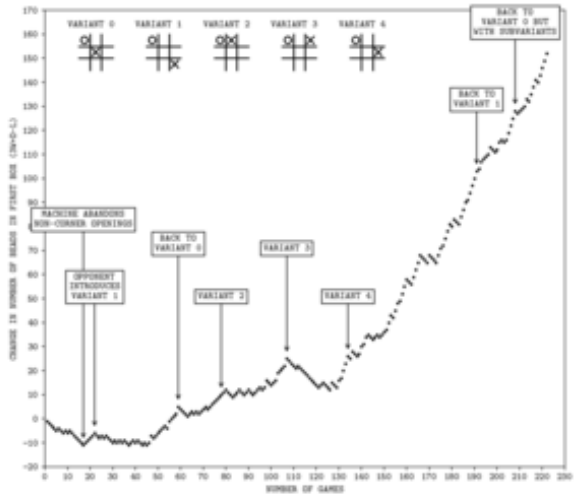


Fig. 1. A scatter graph showing the results of Donald Michie's games against MENACE

Results & Conclusion

The Menace (Matchbox Educable Noughts And Crosses Engine) code we've implemented plays Tic-Tac-Toe and learns from its mistakes. It adjusts its moves based on what works and what doesn't, getting better at the game over time. By changing the number of beads in its matchboxes, Menace can figure out the best moves to make. This shows how a simple idea can lead to smart behavior in a game like Tic-Tac-Toe.

We tried to implement MENACE on Google Colab, it's code can be found

Here

References

[1]https://en.wikipedia.org/wiki/Matchbox_Educable_Noughts_and_Crosses_Engine

[2] https://www.mscroggs.co.uk/blog/19

Lab Assignment 9
n-Armed Bandit

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

You can access the Google Colab notebook here.

I. Introduction

Reinforcement learning is a field of machine learning concerned with decision-making in sequential situations, where an agent interacts with an environment to maximize some notion of cumulative reward.

One classic problem in reinforcement learning is the n-armed bandit problem,

In the n-armed bandit problem, imagine a gambler faced with a row of slot machines (bandits), each with an unknown probability of paying out a reward. The gambler's goal is to figure out which machine(s) to pull to maximize their total reward over time. This problem highlights the tension between exploring different options to gather information and exploiting known good choices to maximize immediate gain.

We have used epsilon-greedy algorithm to to address this exploration-exploitation trade-off.

Keywords: Exploitation - Exploration in simple n-arm bandit reinforcement learning task, epsilon-greedy algorithm

II. Objectives

- Implement an epsilon-greedy algorithm to maximize the expected reward in a binary bandit problem with two rewards: success (1) and failure (0).
- Develop a 10-armed bandit where all ten mean-rewards start out equal and then take independent random walks by adding a normally distributed increment with mean zero and standard deviation 0.01 to all mean-rewards on each time step.
- Develop a modified epsilon-greedy agent capable of effectively handling non-stationary rewards in the 10-armed bandit problem, a scenario where traditional epsilon-greedy algorithms struggle due to the dynamic nature of the reward distributions. Experiment with the modified agent over a minimum of 10,000 time steps

III. Methodology

A. Problem Setup:1

- We address a binary bandit problem where there are two bandits, A and B.
- Each bandit has two actions: 0 and 1, representing different choices.
- Bandit A has success probabilities [0.1, 0.2] for its actions.
- Bandit B has success probabilities [0.8, 0.9] for its actions.

B. Epsilon-Greedy Algorithm:

For 2000 iterations

Algorithm 1 Epsilon-Greedy Algorithm for binary bandit

Initialize Q as an array of zeros of size 2
Initialize N as an array of zeros of size 2
Initialize $actions\_taken$ as an array of zeros of size $N_{\text{iterations}}$
Initialize $avg\_rewards$ as an array of zeros of size $N_{\text{iterations}}$
for $i = 0$ **to** $N_{\text{iterations}} - 1$ **do**
  if random_number() $< e$ **then**
    $A \leftarrow$ Randomly choose an action from [0, 1] {Exploration}
  else
    $A \leftarrow \text{argmax}(Q)$ {Exploitation}
  end if
  $actions\_taken[i] \leftarrow A$
  $R \leftarrow$ binary_bandit_A(A)
  $N[A] \leftarrow N[A] + 1$
  $Q[A] \leftarrow Q[A] + \frac{R - Q[A]}{N[A]}$ {Update Q-value}
  if $i == 0$ **then**
    $avg\_rewards[i] \leftarrow R$
  else
    $avg\_rewards[i] \leftarrow \frac{(i-1) \times avg\_rewards[i-1] + R}{i}$ {Update average reward}
  end if
end for=0

In this algorithm, we're using the epsilon-greedy strategy to tackle the binary bandit problem. We initialize arrays to track rewards, action counts, actions taken, and average rewards. Then, over a set number of iterations, we randomly decide

between exploring new actions and exploiting known ones. Based on this decision, we select an action, observe the reward, and update our estimates accordingly.

For the 10-armed bandit, this function simulates a nonstationary bandit problem by adding random noise to the mean reward estimates of each action.

Algorithm 2 Non-Stationary Bandit Function

Require: Action $action$, Mean reward estimates m (array of size 10)

1: Generate random noise v from a normal distribution with mean 0 and standard deviation 0.01
2: Update mean reward estimates m by adding the noise v to each element
3: Get the value of the chosen action from the updated mean rewards m
4: **return** Value of the chosen action and updated mean reward estimates m =0

We have implemented two functions, one the standard epsilon-greedy function and another one with modified epsilon-greedy, for solving the non-stationary bandit problem

Algorithm 3 Modified Epsilon-Greedy Agent

Require: Exploration rate ϵ, Number of steps $steps$

1: Set decay rate $\alpha = 0.7$
2: **Same as standard Epsilon greedy**
3: **for** $i = 0$ **to** $steps - 1$ **do**
4: **if** random_number() $> \epsilon$ **then**
5: $A \leftarrow \text{argmax}(Q)$
6: **else**
7: $A \leftarrow \text{random\_choice}(\{0, 1, \ldots, 9\})$
8: **end if**
9: $actions\_taken[i] \leftarrow A$
10: $RR, m \leftarrow \text{bandit\_nonstat}(A, m)$
11: $N[A] \leftarrow N[A] + 1$
12: $Q[A] \leftarrow Q[A] + (RR - Q[A]) \times \alpha$ {Update Q-value with alpha}
13: **Update same as standard EG**
14: **end for**=0

Standard Epsilon-Greedy Agent:

- Uses a fixed step-size for updating Q-values.
- Update rule: $Q[A] \leftarrow Q[A] + 0.1 \times (R - Q[A])$.

Modified Epsilon-Greedy Agent:

- Utilizes a decaying step-size for updating Q-values.
- Update rule: $Q[A] \leftarrow Q[A] + (R - Q[A]) \times \alpha$, where α is a decay rate parameter.

The algorithm concludes by providing statistics on rewards, actions, and average rewards over time

IV. Results

Binary Bandit

In fig.1 We can see the actions taken in each iteration for Binary Bandit A. It's likely to show a mix of both actions 0 and 1, with potentially more and more exploitation (choosing the action with the highest estimated reward) later on as the algorithm learns and also exploration at few interval.

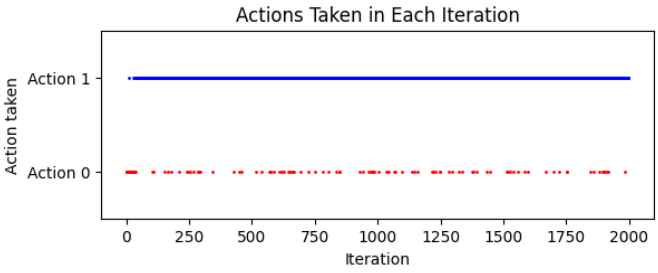


Fig. 1. iterations vs iteration taken in Binary Bandit A

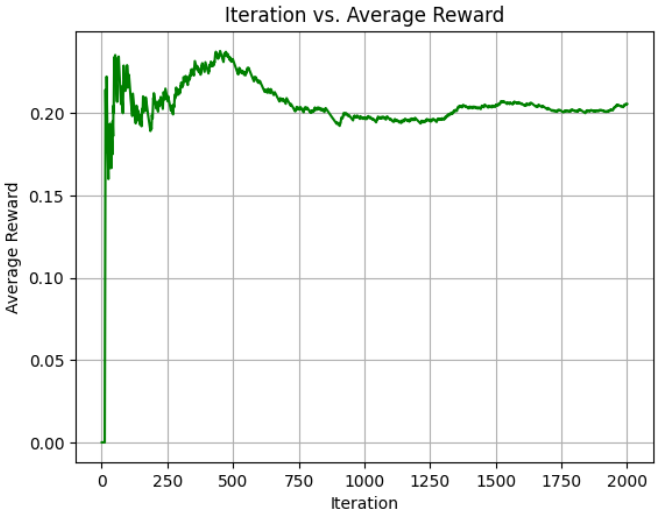


Fig. 2. Average Reward

In Fig.2 We can see the average reward changes over iterations. Initially, the average reward might fluctuate a lot due to exploration, but as the algorithm learns which action is better, the average reward should converge towards the true expected reward of the optimal action.

10 armed Bandit

In the results:

We can see the actions taken in each iteration, In standard epsilon greedy and modified epsilon greedy in Fig.3 and Fig.4 respectively.

Standard Epsilon-Greedy:

- The running average reward starts from an initial value and gradually increases as the algorithm learns which actions yield higher rewards.
- It appears to stabilize around 0.5, indicating that the algorithm is achieving a certain level of performance in terms of average reward as we ccan see in fig.5

Modified Epsilon-Greedy:

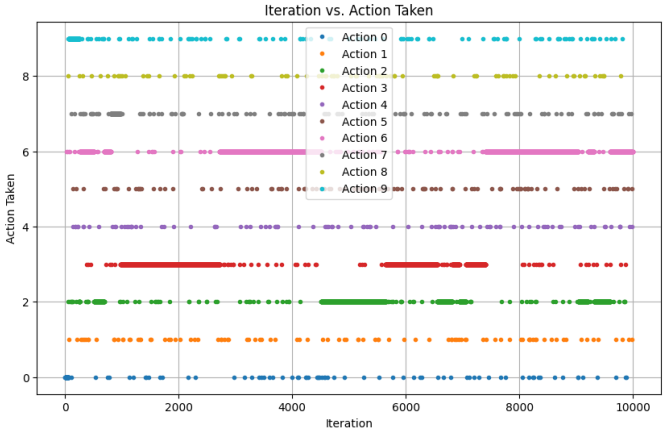


Fig. 3. Actions taken in standard epsilon greedy

Fig. 4. Actions taken in modified epsilon greedy

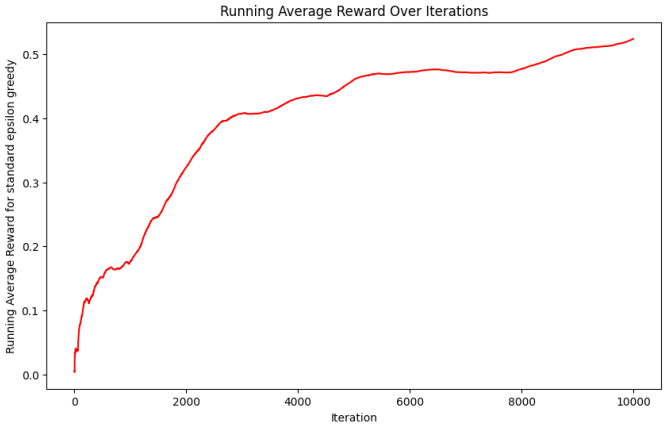


Fig. 5. Average Reward for standard epsilon greedy

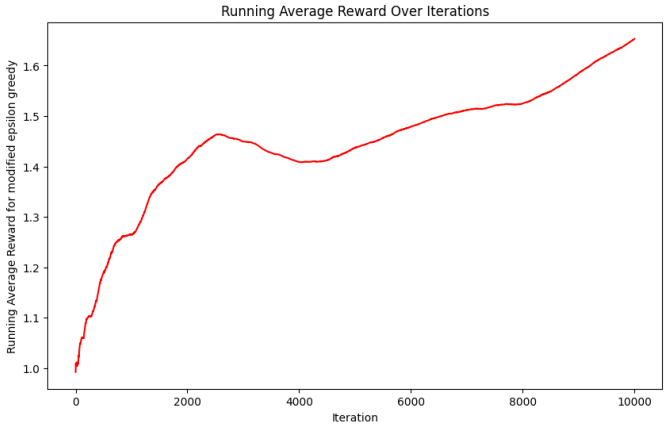


Fig. 6. Average Reward for modified epsilon greedy

- The running average reward also starts from an initial value and increases over iterations.
- However, it seems to stabilize at a higher value, around 1.6, we can see that it fig.6
- This indicates that the modified version of the epsilon-greedy algorithm may be performing better in terms of average reward compared to the standard version.

V. Conclusion

These results suggest that the modification made to the epsilon-greedy algorithm, specifically the introduction of a learning rate (α), has led to improved performance in terms of average reward compared to the standard version. The modified version seems to learn more efficiently or adapt better to the environment, resulting in higher average rewards over time.

References

[1] Reinforcement Learning: an introduction by R Sutton and A Barto (Second Edition)

Lab Assignment 11
Designing a Fuzzy Expert System for Washing Machine Time

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

You can access the Google Colab notebook here.

I. Objective

The objective of this laboratory assignment is to understand and implement a type-1 (Mamdani) fuzzy expert system to determine the appropriate washing time for a load of clothes based on their dirtiness and volume.

II. Introduction

Imagine you're designing a really smart washing machine. You know how sometimes it's hard to guess how long to wash your clothes? Well, with fuzzy logic, it's like giving the washing machine a bit of common sense.

Instead of just having fixed times for washing, you're teaching it to understand two things: how dirty the clothes are and how much laundry there is.

So, let's say the clothes are only a bit dirty and there's not much laundry. In that case, the machine figures it doesn't need a lot of time to wash. But if the clothes are really dirty and the drum is full, it knows it needs more time to get everything clean.

It's kind of like how you'd decide to wash something by hand—you adjust based on how much there is to wash and how dirty it is.

With fuzzy logic, the washing machine can do that all by itself, without you having to worry about picking the exact right time. It's like having a little helper that knows just what to do.

III. Problem Statement

Given the dirtiness level (very dirty, medium dirty, lightly dirty, not dirty) and volume of the load (full load, medium load, low load), the objective is to identify the appropriate washing time (very long time, long time, medium time, little time) using a rule-based approach.

IV. Rule Base

A simple rule base is provided as follows:

Load Volume /Load Dirtiness	Full Load	Medium Load	Low Load
Very Dirty	VLot	VLot	Lot
Medium Dirty	VLot	Lot	MT
Lightly Dirty	Lot	MT	LT
Not Dirty	LT	LT	LT

V. Implementation

- **Dirtiness**, **Volume**, and **Time** are defined as antecedents and consequents, respectively. These are the linguistic variables that will be used in the fuzzy logic system.

Fuzzy Sets for Dirtiness

The variable **dirtiness** has four fuzzy sets defined:

- **not_dirty**: Represents when something is not dirty. It has a triangular membership function with values ranging from 0 to 3 on the dirtiness scale.
- **lightly_dirty**: values ranging from 0 to 6
- **medium_dirty**: values ranging from 3 to 9
- **very_dirty**: values ranging from 6 to 10

Fuzzy Sets for Volume

The variable **volume** has three fuzzy sets defined:

- **low_load**: Represents a low volume or load. It has a triangular membership function with values ranging from 0 to 5 on the volume scale.
- **medium_load**: values ranging from 0 to 10 on the volume scale.
- **full_load**: values ranging from 5 to 10 on the volume scale.

Fuzzy Sets for Time

The variable **time** has four fuzzy sets defined:

- **little_time**: Represents a short amount of time. It has a triangular membership function with values ranging from 0 to 4 on the time scale.
- **medium_time**: 0 to 8 on the time scale.
- **long_time**: 4 to 10
- **very_long_time**: values ranging from 8 to 10

Implementation Explanation

- Each rule consists of an antecedent (input conditions) and a consequent (output action).
- There are 12 rules defined in total, covering different combinations of dirtiness and volume.
- For example, `rule1` states that if the volume is `full_load` and the dirtiness is `very_dirty`, then the washing time should be `very_long_time`.

The control system is created using the ctrl.ControlSystem class, where the rules are passed as a list. In this case, the rules include conditions and actions based on the dirtiness and volume of the load.

And in Simulation, simulation is created using ctrl.ControlSystemSimulation. This simulation is what will actually process inputs and generate outputs based on the defined rules.

Results

The fuzzy logic system was evaluated with three different scenarios:

Scenario 1: Low Dirtiness, High Volume

- Dirtiness Level: 1
- Volume Level: 10
- Washing Time: 3.11

Scenario 2: Medium Dirtiness, Medium Volume

- Dirtiness Level: 4
- Volume Level: 6
- Washing Time: 5.10

Scenario 3: High Dirtiness, Low Volume

- Dirtiness Level: 9
- Volume Level: 6
- Washing Time: 9.10

The fuzzy logic system successfully determined the washing time for each scenario based on the levels of dirtiness and volume.

- Scenario 1: resulted in a relatively short washing time.
- Scenario 2:moderate washing time.
- Scenario 3: the longest washing time.

These results demonstrate the effectiveness of the fuzzy logic system in adjusting washing time according to varying levels of dirtiness and volume, contributing to more efficient and adaptive washing machine operation.

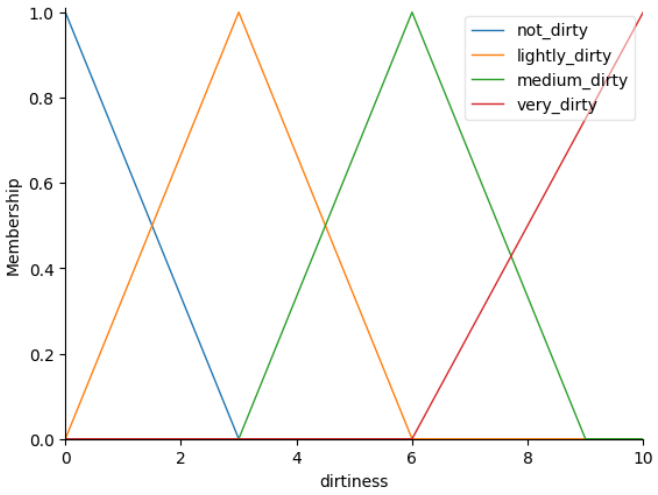


Fig. 1.

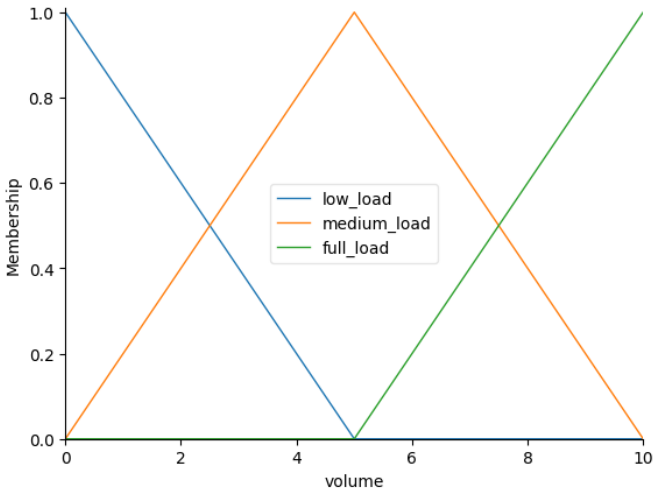


Fig. 2.

Visualization of Fuzzy Sets

Each plot shows the membership function curves for the fuzzy sets defined for the respective variable. The x-axis represents the range of values for the variable (e.g., dirtiness level, volume level, or washing time), while the y-axis represents the degree of membership in each fuzzy set.

The curves depict how input values are mapped to the linguistic terms through fuzzy membership degrees. A higher membership degree indicates a stronger association with that linguistic term.

The visualization helps understand how input values are interpreted and mapped to linguistic terms in the fuzzy logic system.

For example, you can see the transition between different dirtiness levels in Fig.1 like not_dirty, lightly_dirty,

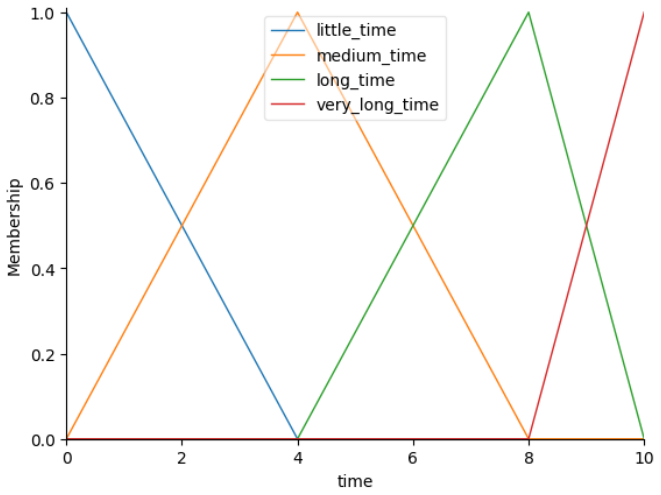


Fig. 3.

medium_dirty, very_dirty and volume levels low_load, medium_load, full_load in Fig.2 based on the shapes of the membership function curves.

Understanding these curves can help in interpreting the output of the fuzzy logic system and how input variables contribute to the final washing time recommendation.

VI. Conclusion

In this laboratory assignment, we successfully designed and implemented a type-1 fuzzy expert system for determining washing time based on dirtiness and volume of the load. This exercise provided insights into the practical application of fuzzy logic in real-world systems.