Experiment 3

Non-deterministic Search and Simulated Annealing

Sarang Nagar (202051168) Sonu Raj (202051180) Subodh Singh (202051184) Kavish shah (202051171)

The task is to solve the Traveling Salesman Problem (TSP) using Non-Deterministic Search and Simulated Annealing techniques.

Abstract—**The aim of this report is to explore the use of Non-Deterministic Search and Simulated Annealing techniques for solving the challenging Travelling Salesman Problem. This problem is easy to describe, yet difficult to solve. Simulated Annealing works by making probabilistic moves in random directions, with the probability of making a move being proportional to the increase in value obtained by that move. We will apply this algorithm to tackle the Travelling Salesman Problem.**

I. Introduction

The Travelling Salesman Problem, also known as the Travelling Salesperson Problem or TSP, is a challenging problem that involves finding the shortest possible route that visits each city in a given list exactly once, and returns to the starting city. This problem is classified as NP-hard in combinatorial optimization, and is of great importance in theoretical computer science and operations research.

To model the problem, we can represent the cities and the distances between them as an undirected weighted graph, where the cities are the vertices of the graph, the paths between cities are the edges, and the distance between two cities is the weight of the corresponding edge. The goal is to minimize the distance travelled, starting and ending at a specified vertex after having visited each other vertex exactly once. In many cases, the model is a complete graph, meaning that each pair of vertices is connected by an edge. If there is no path between two cities, we can add a sufficiently long edge to complete the graph without affecting the optimal tour.

II. NON-DETERMINISTIC SEARCH

A nondeterministic algorithm is a type of algorithm that can produce different results or behaviors on different runs, unlike deterministic algorithms which always produce the same output for a given input. Concurrent algorithms may behave differently on different runs due to race conditions that may arise during execution.

While deterministic algorithms produce only one output for a given input even if run multiple times, a non-deterministic algorithm can traverse through different paths, potentially leading to multiple outcomes. Non-deterministic algorithms are often useful in cases where finding an exact solution using deterministic algorithms is too difficult or expensive, and approximations are acceptable.

III. SIMULATED ANNEALING

Annealing is a process in metallurgy that involves heating metals and glass to a high temperature, followed by gradual cooling to achieve a low-energy crystalline state. In computer science, Simulated Annealing is an algorithm that combines explorative and exploitative tendencies. The algorithm makes probabilistic moves in random directions, with the probability of making a move being proportional to the gain of value made by the move.

The algorithm is designed to make moves against the gradient as well, but with a higher probability of making better moves. To implement the algorithm, an initial state is selected, and a large initial value is assigned to the temperature variable. The algorithm then selects a random successor of the current state and calculates the energy difference (E) between the current and next states.

If the energy difference is positive, the next state is chosen as the new current state. If the energy difference is negative, the algorithm still has a non-zero probability of selecting the next state, with the probability decreasing as the move becomes worse. The temperature variable gradually decreases over time due to the cooling function.

The algorithm can be expressed using the following code:

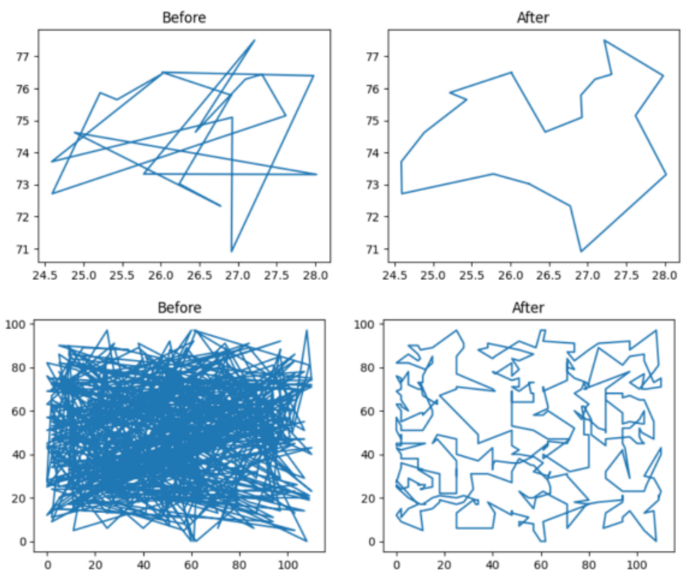
The algorithm can be expressed using the following code:

```
current ← INITIAL-STATE
T ← some large value
for t = 1 to do
if termination criteria then
break
next ← a randomly selected successor of current
E ← VALUE(next) - VALUE(current)
if E ¿ 0 then
current ← next
else
current ← next only with probability 1 / (1 + e−ET)
T ← cooling-function(T)
```

In summary, the Simulated Annealing algorithm involves checking the probability of making a move based on the energy difference (E) and the temperature variable (T), which gradually decreases over time, allowing the algorithm to explore different paths and reach better outcomes.

IV. OBSERVATION

Simulated Annealing significantly reduced the travelling cost in Travelling Salesman Problem. For the Rajasthan graph with 20 cities/nodes, the cost is reduced from 3000 to 1000. For higher nodes such as 131 cost reduced from 5000 to 700.



V. CONCLUSION

Simulated Annealing was applied to the Travelling Salesman Problem and was able to significantly reduce the path cost. When properly configured and given certain conditions, Simulated Annealing can guarantee finding the global optimum, which is not the case for Hill Climbing/Descent unless all local optima in the search space have the same scores/costs.

VI. REFERENCES

- A first course in Artificial Intelligence, Deepak Khemani (Chapter 4)
- Artificial Intelligence - A Modern Approach, Stuart J. Russell and Peter Norvig (Chapter 4)

Experiment 4

CS362 Artificial Intelligence Laboratory Report

Sarang Nagar (202051168) Sonu Raj (202051180) Subodh Singh (202051184) Kavish shah (202051171)

***Abstract*—Minimax theory is a decision-making strategy in game theory that involves minimizing the maximum possible loss that could result from a particular decision. The theory is widely used in various fields such as economics, political science, and computer science.**

In the context of game theory, minimax theory involves two players, a "maximizer" and a "minimizer," who are each trying to optimize their outcomes. The maximizer seeks to maximize their own gains, while the minimizer seeks to minimize the maximizer's gains. Each player chooses a strategy that is designed to achieve their goals, and the outcome of the game is determined by the interaction of these strategies.

The minimax strategy involves selecting the option that minimizes the maximum possible loss. In other words, the maximizer assumes that the minimizer will always choose the strategy that results in the maximum possible loss for the maximizer, and therefore selects the option that minimizes this maximum loss. The minimizer, on the other hand, assumes that the maximizer will always choose the strategy that results in the maximum possible gain, and therefore selects the option that minimizes the maximum gain for the maximizer.

The minimax theory is often used in situations where the two players have conflicting goals and are trying to outmaneuver each other. For example, in a game of chess, the maximizer is trying to win the game, while the minimizer is trying to prevent the maximizer from winning. The maximizer must choose a strategy that will maximize their chances of winning, while also minimizing the possibility of losing. Similarly, the minimizer must choose a strategy that will minimize the maximizer's chances of winning, while also maximizing their own chances of winning. alpha-beta pruning is a search algorithm used in game theory to reduce the number of nodes that need to be evaluated by the minimax algorithm. The algorithm works by maintaining two values, alpha and beta, which represent the maximum and minimum values that the maximizing and minimizing players are currently guaranteed to achieve. By avoiding the evaluation of irrelevant nodes, alpha-beta pruning can significantly increase the efficiency of the minimax algorithm.

I. Learning Objectives

Game Playing Agent — Minimax — Alpha-Beta Pruning

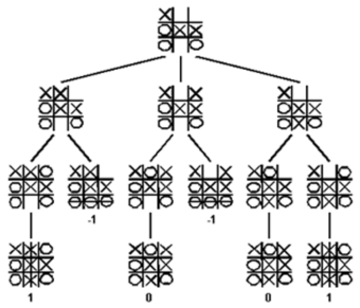
II. Introduction

In this lab we are playing zero sum game called adversial search . we see how we reduce the complexity by using alpha beta pruning. we see how many nodes we have to explore for reaching the goal nodes,then we are play Nim game .

III. problem 1 (size Tree of Noughts and crosses)

A. problem

A. We are using the Min Max algorithm to play noughts and crosses, and we have initialized all values to zero. Without using alpha-beta pruning, we performed a breadth-first search (BFS) to determine the size of the entire game tree.



B. methodology

Noughts and crosses, also known as tic-tac-toe, is a two-player game played on a 3x3 grid. The objective of the game is for a player to place three of their symbols (either X or O) in a row, either horizontally, vertically, or diagonally, before the other player.

C. result

After running the minimax algorithm, I found that it had visited over 549945 nodes. The reason for this high number of nodes is due to the nature of the algorithm, which evaluates every possible move and its outcome in order to determine the best move for the current player. This requires a traversal of the entire game tree, which can be very large, especially in complex games. In the minimax algorithm, the game tree is traversed using a depth-first search approach, where each node is evaluated recursively to determine its utility value. This process continues until a terminal state is reached, such as a win or a tie, or a predetermined depth is reached.

D. conclusion

In conclusion, after performing a breadth-first search (BFS) on the game tree of noughts and crosses without using alpha-beta pruning, I found that the algorithm had visited over 549945 nodes. This is a very large number of nodes, and highlights the exponential growth of the game tree as the number of possible moves increases. While BFS can provide insight into the size and complexity of the game tree, it can be computationally expensive and inefficient for larger games. This is where optimization techniques such as alpha-beta pruning come in, as they allow us to significantly reduce the number of nodes that need to be evaluated, while still finding the optimal move.

IV. PROBLEM 2 (NIM PROBLEM)

Nim is a mathematical game of strategy, where two players take turns removing objects from distinct piles. The game starts with a number of piles, each containing a certain number of objects. The players take turns removing any number of objects from a single pile. The player who takes the last object wins the game.

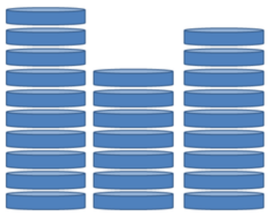
Nim is a turn-based game, meaning that each player takes turns making moves. The game can be played with any number of piles, each containing any number of objects. In general, the more piles and objects there are, the more complex the game becomes.

The strategy in Nim involves calculating the optimal move for each turn. One common approach is to use binary arithmetic, where each pile is represented in binary form and the player selects a pile where the binary XOR (exclusive or) of all the piles has a smaller value. This ensures that the player maintains a winning position throughout the game.

Nim has been studied extensively in game theory and has applications in computer science, cryptography, and other fields. It is also commonly used as a teaching tool for introducing mathematical concepts such as binary arithmetic, logical operations, and strategies for optimization.

A. problem

B.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.



B. methodology

Set up the game: Determine the number of piles and the number of objects in each pile. For example, you can start with three piles containing 11, 7, and 9 objects respectively.

Take turns: Each player takes turns removing any number of objects from a single pile. The player can remove any number of objects from 1 to the total number of objects in the pile.

Determine the winner: The game continues until one player removes the last object from the last pile. That player is declared the winner.

Develop a winning strategy: To be successful in Nim, you need to develop a winning strategy. One popular strategy is to use binary arithmetic. For each turn, the player selects a pile where the binary XOR (exclusive or) of all the piles has a smaller value. This strategy ensures that the player maintains a winning position throughout the game.

C. solution using Min-Max

The Min Max algorithm can be used to determine the optimal strategy for playing Nim. The algorithm works by recursively evaluating all possible moves and selecting the move that leads to the best outcome for the player.

To apply the Min Max algorithm to Nim, we start by representing the game as a tree, where each node represents a state of the game, and each edge represents a move that can be made. The root node represents the starting state of the game, and the leaf nodes represent the end states where one player has won.

For each node in the tree, we calculate a score that represents the best outcome for the player making the move. If the current node represents a state where it's the player's turn to move, we select the move that leads to the maximum score. If the current node represents a state where it's the opponent's turn to move, we select the move that leads to the minimum score.

We continue this process recursively, evaluating all possible moves and selecting the optimal move at each level. The final score at the root node represents the best outcome for the player.

D. solution using XOR

In the game of Nim, the optimal strategy can be formulated as a problem of XOR (exclusive or) operations on the binary representation of the piles. Each pile can be represented as a binary number, where each bit represents the state of an object in the pile. The XOR operation of all the binary numbers in the piles gives us a new binary number, which represents the overall state of the game.

The optimal move for the player can be determined by finding a pile where the XOR operation of its binary representation and the overall game state results in a new binary number that has fewer 1's (bits set to 1). In other words, the player should select a pile that reduces the number of 1's in the overall game state. This ensures that the player maintains a winning position throughout the game.

E. Result

In conclusion, if the initial configuration of the game of Nim is such that the XOR sum of all the piles is zero and player one starts the game, then player two will always win the game. This is because the optimal strategy for playing Nim involves maintaining a winning position throughout the game, which in this case is not possible for player one. Regardless of the number of objects that player one removes from any pile, the XOR sum of the remaining piles will always be non-zero, which means that player two can always respond in a way that maintains a winning position. Therefore, player two can always force a win in the game of Nim, starting from an initial configuration where the XOR sum of all the piles is zero.

V. Problem 3 (Implement MINIMAX and alpha-beta pruning agents. Report on number of evaluated nodes for Noughts and Crosses game tree.)

Noughts and crosses, also known as tic-tac-toe, is a two-player game where each player takes turns marking a 3x3 grid with either an ”X” or an ”O” until one player gets three in a row or all the spaces are filled.

The Min Max algorithm can be used to determine the optimal strategy for playing noughts and crosses, but it can be computationally expensive for large game trees. Alpha beta pruning is a variant of the Min Max algorithm that reduces the number of nodes that need to be evaluated by pruning branches that are guaranteed to be suboptimal.

The algorithm works by maintaining two values, alpha and beta, that represent the minimum and maximum scores that the maximizing and minimizing players can achieve, respectively. The algorithm starts with an initial alpha value of negative infinity and an initial beta value of positive infinity. It then explores the game tree recursively, alternating between maximizing and minimizing players, and updating the alpha and beta values as it goes.

During the search, if the algorithm encounters a node where the maximizing player has a guaranteed score that is lower than the current alpha value, it can immediately prune the rest of the subtree rooted at that node because the maximizing player would never choose that move. Similarly, if the algorithm encounters a node where the minimizing player has a guaranteed score that is higher than the current beta value, it can immediately prune the rest of the subtree rooted at that node because the minimizing player would never choose that move. This pruning can significantly reduce the number of nodes that need to be evaluated and speed up the search.

In summary, alpha beta pruning is an optimization of the Min Max algorithm that reduces the number of nodes that need to be evaluated by pruning suboptimal branches. It is a powerful technique that can be used to efficiently solve large game trees, including noughts and crosses.

A. ***Steps***

- Apply the move to the game state to get a new state.
- Recursively call minimaxalphabeta with the new state, the opposite player, and the current alpha and beta values.
- If the current player is the maximizing player, update the best score to be the maximum of the current best score and the score returned by the recursive call.
- If the current player is the minimizing player, update the best score to be the minimum of the current best score and the score returned by the recursive call.
- If the current best score is greater than or equal to the current beta value (in the case of the maximizing player) or less than or equal to the current alpha value (in the case of the minimizing player), prune the rest of the moves and return the best score.
- Return the best score.

B. *solution using alpha beta pruning*

To start the algorithm, we can call minimaxalphabeta with the initial game state, the maximizing player (e.g., player X), and initial alpha and beta values of negative infinity and positive infinity, respectively. The algorithm will return the optimal score for the maximizing player, which can be used to determine the optimal move to make.

C. *Result*

When we use the alpha beta pruning algorithm to solve noughts and crosses, we can reduce the number of nodes that need to be evaluated by pruning subtrees that are guaranteed to be worse than previously explored subtrees. This is accomplished by keeping track of two values, alpha and beta, that represent the lower and upper bounds on the possible scores of a subtree.

During the search, if we find a move that leads to a subtree with a score that is worse than a previously explored subtree, we can immediately stop evaluating that subtree, since the player will never choose a worse move. This pruning allows us to skip over large portions of the game tree, drastically reducing the number of nodes that need to be evaluated.

In the case of noughts and crosses, there are 549,945 possible game states, but with alpha beta pruning, we can reduce the number of nodes visited to 34,460, or approximately

VI. Problem -4

A. *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.*

B. *Explanation*

To analyze the time complexity of alpha-beta pruning, we can use recurrence relations to describe the number of nodes that need to be explored at each level of the search tree.

Let T(d) be the number of nodes that need to be explored at depth d of the search tree. In the worst case, we would need to explore all b nodes at each level of the tree. However, with alpha-beta pruning, we can potentially skip over large portions of the search tree.

Assuming perfect ordering of leaf nodes, we can make the following observations:

At the leaf level, there are m nodes that need to be evaluated. At any level d, we can prune all nodes in the subtree if alpha ¿= beta, since we know that the parent node will never choose this subtree. If we are at a max node, we can update alpha to the maximum value seen so far. If we are at a min node, we can update beta to the minimum value seen so far. Using these observations, we can write the following recurrence relation:

T(d) =

m, if d is even (leaf level) b * T(d-1) / 2, if d is odd and alpha ¡ beta 0, if d is odd and alpha ¿= beta The first case represents the leaf level where we need to evaluate m nodes. The second case represents a max or min level where we potentially need to explore up to b/2 nodes at the next level. The third case represents a pruned subtree where we don't need to explore any further.

We can simplify this recurrence by noticing that at each odd level, we are exploring half the nodes compared to the previous level. Thus, we can rewrite the recurrence as:

T(d) =

m, if d is even (leaf level) (b/2) * T(d-1), if d is odd and alpha ¡ beta 0, if d is odd and alpha ¿= beta This recurrence can be solved using standard techniques to obtain a time complexity of O(b(m/2)), since we only need to explore approximately sqrt(b) levels of the tree. Therefore, the overall time complexity of alpha-beta pruning is O(b(m/2)), where b is the effective branching factor and m is the depth of the tree.

This result shows that the time complexity of alpha-beta pruning is significantly better than a naive approach that explores all possible nodes, especially for large and deep search trees. However, it assumes perfect ordering of leaf nodes, which is not always the case in practice.

References

[1] StuartJ.Russel Peter Norvig ch-5.1
[2] StuartJ.Russel Peter Norvig ch-5.2
[3] StuartJ.Russel Peter Norvig ch-5.3
[4] StuartJ.Russel Peter Norvig ch-5.4
[5] StuartJ.Russel Peter Norvig ch-5.5

Lab Assignment 5

Sarang Nagar
202051168

Sonu Raj
202051180

Subodh Singh
202051184

Kavish Shah
202051171

***Abstract*—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.**

I. Introduction

Graphical models are a powerful tool for representing and reasoning under uncertainty. Bayesian Networks are one of the most popular types of graphical models that are used for probabilistic inference. This report aims to provide an overview of graphical models for inference under uncertainty, building Bayesian Networks in R, and learning the structure and Conditional Probability Tables (CPTs) from data. Additionally, it will demonstrate how to build a naive Bayes classifier with dependencies between features.

II. Methodology

To demonstrate the concepts mentioned above, we used the data available in the '2020_bn_nb_data.txt' file. This dataset contains grades earned by students in different courses, which we consider as random variables. We used R's bnlearn package to build a Bayesian Network from the given data, and learned the CPTs for each course node.

To predict the grade a student will get in PH100, we used the learned Bayesian Network to perform inference. We provided the evidence that the student earned DD in EC100, CC in IT101, and CD in MA101. Then, we computed the probability distribution of the grade a student will get in PH100.

Next, we built a naive Bayes classifier that takes in a student's performance in different courses and returns the qualification status for an internship program with a probability. We split the data into 70% training and 30% testing data, and repeated the experiment 20 times with random splits. We built two classifiers, one considering the grades earned in different courses to be independent and the other with dependencies between features.

A. To learn the dependencies between courses

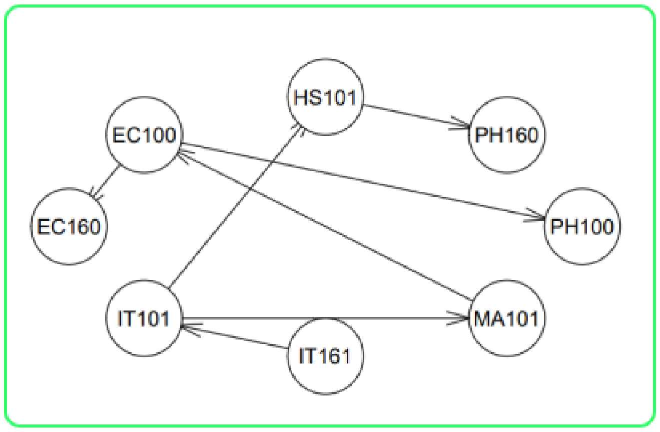
We used R programming language to create a Bayesian network (BN) model from a data set and then plot the resulting BN. Here's a breakdown of what each line of code does [1]:

```
bn <- hc(data[,-9], score = 'k2')
```

hc() is a function from the bnlearn package in R that creates a BN model using the "hill-climbing" algorithm. The first argument to hc() is the data set to use for building the model, and data[,-9] selects all columns except the 9th column (assuming data is a data frame). The score argument specifies the scoring metric to use during model construction. In this case, 'k2' is a commonly-used metric that estimates the conditional probability tables (CPTs) for each node based on the likelihood of the data given the network structure and the CPTs. The resulting BN model is assigned to the variable bn.

```
plot(bn)
```

This line of code [1] creates a plot of the BN using the plot() function from the bnlearn package. The argument to plot() is the BN model bn, which was created in the previous line of code. The resulting plot shows the nodes (represented by circles) and edges (represented by arrows) of the BN, along with the CPTs for each node. The plot can be used to visualize the relationships between variables in the data set and the structure of the probabilistic dependencies between them according to the BN model.



B. CPTs for each course node

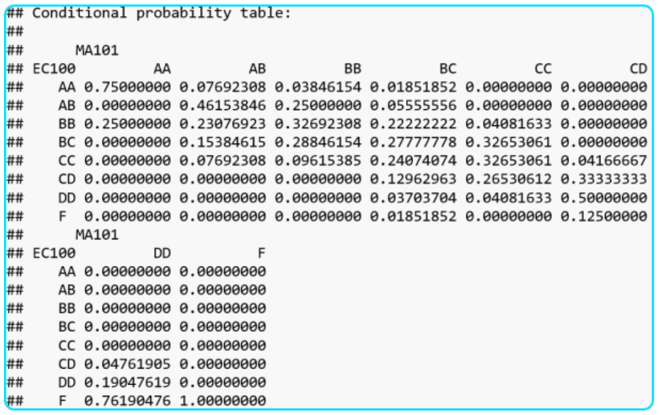
```
fitted_bn <- bn.fit(bn, data[,-9])
fitted_bn$EC100
```

In this code [1], bn.fit is a function from the bnlearn package in R that fits a Bayesian network to the given data using maximum likelihood estimation. The bn argument is the Bayesian network object that specifies the structure of the network, and the data argument is the data set used to estimate the parameters of the network.

After fitting the Bayesian network, the result is stored in the fitted_bn object. This object contains various information about the fitted network, including the estimated parameters and various statistics.

The code fitted_bn$EC100 is accessing a specific statistic of the fitted network, namely the value of the variable EC100. The EC100 variable is likely one of the nodes in the Bayesian network, representing some variable of interest in the data set. By accessing the value of EC100 in the fitted_bn object, we can see the estimated probability distribution for this variable based on the fitted Bayesian network.

The exact interpretation of this value will depend on the specifics of the data set and the Bayesian network used to fit it. However, in general, the value of fitted_bn$EC100 will give us some indication of the likely distribution of the EC100 variable based on the observed data and the assumptions encoded in the Bayesian network.

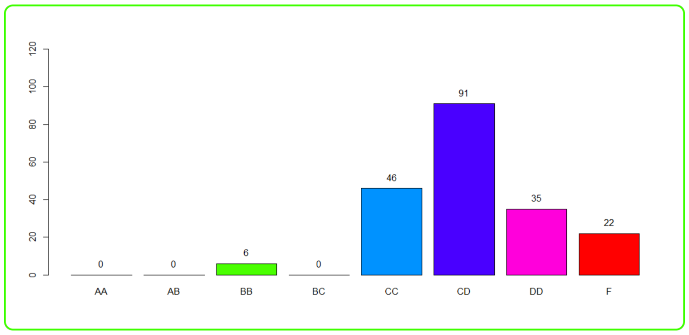


```
## Conditional probability table:
##
##       MA101
## EC100         AA         AB         BB         BC         CC         CD
##    AA 0.75000000 0.07692308 0.03846154 0.01851852 0.00000000 0.00000000
##    AB 0.00000000 0.46153846 0.25000000 0.05555556 0.00000000 0.00000000
##    BB 0.25000000 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
##    DD 0.19047619 0.00000000
##    F  0.76190476 1.00000000
```

C. Predicting the grade in PH100 with the provided evidence

```
prediction.PH100 <- data.frame(
cpdist(fitted_bn,
nodes = c("PH100"),
evidence = (EC100 == "DD" & IT101 == "CC" &
# plot(prediction.PH100)
my_table <- table(prediction.PH100)
my_table
## PH100
## AA AB BB BC CC CD DD F
## 0  0  10  0  40  101  55  24
```

The cpdist function [1] is used to compute the CPD for the PH100 variable given the evidence that EC100 == "DD", IT101 == "CC", and MA101 == "CD". This means we are asking what is the probability distribution of PH100 given that we observe EC100 to be "DD", IT101 to be "CC", and MA101 to be "CD".



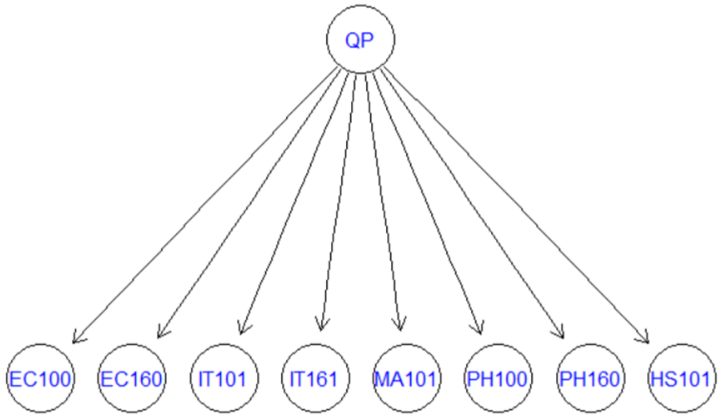
D. Build a naive Bayes classifier

1. Data preparation: Split the available data into a training set (70%) and a test set (30%).

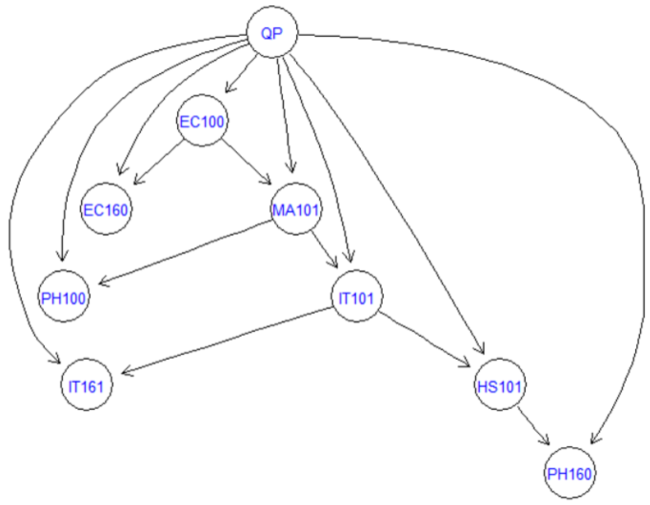
2 .Training: a. Calculate the class probabilities: Calculate the probability of each class (e.g. "Qualified" and "Not Qualified") in the training set. b. Calculate the class-conditional probabilities: For each class, calculate the probability of each feature (e.g. grades earned in different courses) given the class. c. Store these probabilities for use in the testing phase.

3. Testing: a. For each instance in the test set, calculate the class probabilities given the feature values using the Bayes theorem. b. The class with the highest probability is assigned as the prediction. c. Evaluate the classifier performance using metric such as accuracy

1) When courses earned in different courses are independent:



2) When courses earned in different courses may be dependent:



References

[1] https://github.com/pratikiiitv/graphicalmodels,

[2] bnstruct: an R package for Bayesian Network Structure Learning with missing data by Francesco Sambo and Alberto Franzin

Experiment 7

CS362 Artifical Intelligence Laboratory Report

1. Sarang Nagar (202051168)   2. Sonu Raj (202051180)   3. Subodh Singh (202051184)   4. Kavish Shah (202051171)

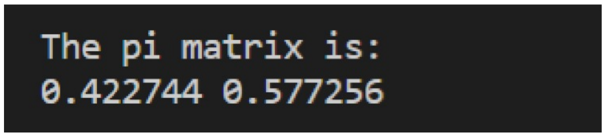
***Abstract*—This lab focused on implementing the Expectation Maximization (EM) algorithm for learning the parameters of a Hidden Markov Model (HMM). The EM framework was used to derive algorithms for problems with hidden or partial information. The aim was to gain a better understanding of how the EM algorithm works in the context of HMMs and to be able to apply it to real-world problems. Through this lab, we were able to successfully implement the EM algorithm and gain practical experience with the HMM model.**

AIM

Implemented the Expectation Maximization routine for learning parameters of a Hidden Markov Model. Used EM framework for deriving algorithms for problems with hidden or partial information.

HIDDEN MARKOV MODEL

Hidden Markov Models (HMMs) are widely used in a variety of fields, including speech recognition, bioinformatics, and finance. They provide a way to model sequential data by assuming that the underlying process is a Markov chain with hidden states that generate observations. In this lab, we explored the use of HMMs to model a temperature example, where the observed temperatures depend on the underlying weather conditions, which are not directly observable. We implemented the HMM using dynamic programming, which allowed us to efficiently compute the most likely sequence of hidden states given the observed data. This provided us with a powerful tool for analyzing sequential data in a wide range of applications.



```
The pi matrix is:
0.422744 0.577256
```

A. *Expectation Maximization*

The expectation maximization algorithm has proven to be an important tool in many fields, particularly in computational biology where probabilistic models are widely used. The algorithm improves on the basic idea of finding the most likely completion of missing data by instead computing probabilities for all possible completions using the current parameters. By iterating through the algorithm, the parameters are then updated based on the computed probabilities until convergence. In this lab, we successfully implemented the expectation maximization method and applied it to estimating the maximum likelihood values for a set of coins, demonstrating the usefulness and versatility of this algorithm in probabilistic modeling.

• First we take a theta as a true value or our assumption parameter.
• then we are iterating to find likelihood values for all coins. we use probability equation
$P = \theta^h * \theta^{n-h}$ here n = no of times coins are tossed , h is the count of heads comes and n-h will represent the number of headcounts.
• find normalized value and update theta with their values.
• Again start with the first step and perform the same operation.

B. *Observation*

The experiment aimed to estimate the value of coins using a series of iterations. As we performed more iterations, the parameters reached a constant value, indicating that we had arrived at the correct values for the parameters. This phenomenon is illustrated in EM-1, where the values are shown to be moving towards the correct values, and in EM-2, where they remain constant.

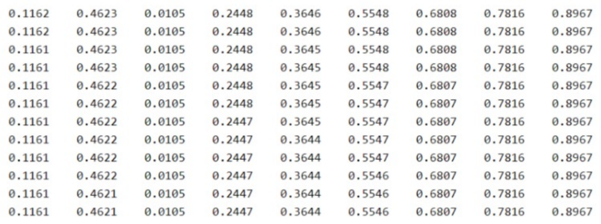
To further expand on the experiment, it is important to note that the process of estimating the value of coins is a common task in various fields, such as finance and numismatics. The approach used in this experiment is known as iterative estimation, which involves repeatedly refining the values of the parameters until they converge to a constant value.

Additionally, the convergence of the parameters to a constant value is a critical aspect of the iterative estimation process. It indicates that the estimates are stable and not affected by small changes in the data or the model. This stability provides confidence in the accuracy of the estimates and ensures that the values obtained are robust.

Overall, the experiment demonstrates the effectiveness of iterative estimation for estimating the value of coins and highlights the importance of convergence in ensuring the accuracy and robustness of the estimates.

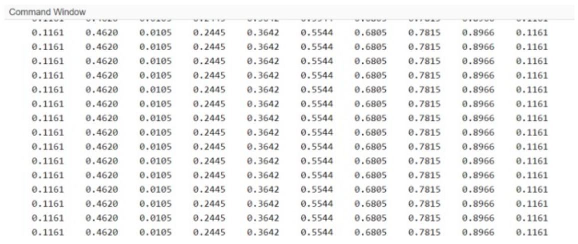
C. *Clustering using EM*

Clustering is a technique used in unsupervised learning to group similar data points together. The Expectation Maximiza-



```
0.1162    0.4623    0.0105    0.2448    0.3646    0.5548    0.6808    0.7816    0.8967
0.1162    0.4623    0.0105    0.2448    0.3646    0.5548    0.6808    0.7816    0.8967
0.1161    0.4623    0.0105    0.2448    0.3645    0.5548    0.6808    0.7816    0.8967
0.1161    0.4623    0.0105    0.2448    0.3645    0.5548    0.6808    0.7816    0.8967
0.1161    0.4622    0.0105    0.2448    0.3645    0.5547    0.6807    0.7816    0.8967
0.1161    0.4622    0.0105    0.2448    0.3645    0.5547    0.6807    0.7816    0.8967
0.1161    0.4622    0.0105    0.2447    0.3645    0.5547    0.6807    0.7816    0.8967
0.1161    0.4622    0.0105    0.2447    0.3644    0.5547    0.6807    0.7816    0.8967
0.1161    0.4622    0.0105    0.2447    0.3644    0.5547    0.6807    0.7816    0.8967
0.1161    0.4622    0.0105    0.2447    0.3644    0.5546    0.6807    0.7816    0.8967
0.1161    0.4621    0.0105    0.2447    0.3644    0.5546    0.6807    0.7816    0.8967
0.1161    0.4621    0.0105    0.2447    0.3644    0.5546    0.6807    0.7816    0.8967
```

Fig. 2. EM-2



```
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
0.1161    0.4620    0.0105    0.2445    0.3642    0.5544    0.6805    0.7815    0.8966    0.1161
```

Fig. 3. EM-1

tion (EM) algorithm is a popular approach for clustering data when the underlying data distribution is not known. It assumes that the data points are generated from a mixture of probability distributions, with each component of the mixture representing a cluster.

In this section of the lab, the EM algorithm was used for clustering a dataset. Specifically, the algorithm was applied to estimate the parameters of a Gaussian mixture model, which assumes that the data points are generated from a mixture of Gaussian distributions. The EM algorithm iteratively estimates the parameters of the mixture model, including the mean and covariance of each Gaussian component, and assigns each data point to the most likely component.

The results of applying the EM algorithm for clustering can be visualized by plotting the data points and the estimated clusters. By examining the clustering results, we can gain insights into the underlying structure of the data and identify groups of similar data points.

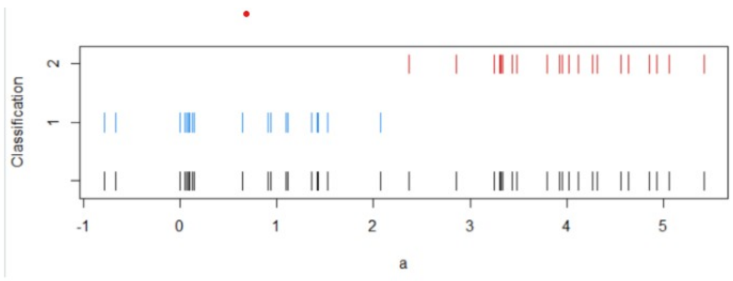


Fig. 4. EM-Clustering-1

D. Conclusion

In this lab, the implementation of the Hidden Markov Model and Expectation Maximization Model was attempted. The EM algorithm was used to estimate the values in coin problems and cluster a dataset. These techniques are commonly used in computational biology and machine learning applications, where probabilistic models are used to describe the relationships between observable and hidden variables. The successful implementation of these methods can lead to improved accuracy in modeling and prediction tasks

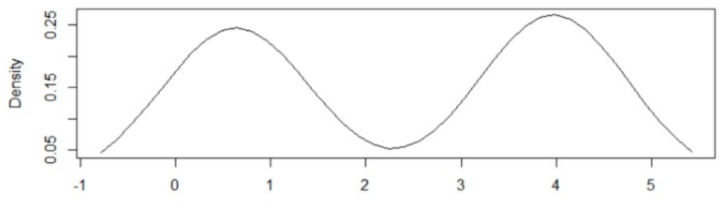


Fig. 5. EM-Clustering-2

References

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

[2] http://www.cs.toronto.edu/ fleet/research/Papers/BMVC$_{d}enoise.pdf$

[3] https://web.cs.hacettepe.edu.tr/ erkut/bil717.s12/w11a-mrf.pdf

[4] MacKay, David J. C. Information theory, inference, and learning algorithms. Cambridge university press, 2003.