VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

NGUYỄN ĐÌNH VIỆT HOÀNG – 522H0120

NGUYỄN NHẬT CHIÊU - 522H0133

ĐẶNG CÔNG MINH - 522H0095

TRẦN THIÊN ÂN - 522H0165

VÕ MINH TÀI - 522H0168

FINAL REPORT INTRODUCTION TO ARTIFICIAL INTELLIGENCE

HO CHI MINH CITY, YEAR 2024

VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

NGUYỄN ĐÌNH VIỆT HOÀNG – 522H0120

NGUYỄN NHẬT CHIÊU - 522H0133

ĐẶNG CÔNG MINH - 522H0095

TRẦN THIÊN ÂN - 522H0165

VÕ MINH TÀI - 522H0168

FINAL REPORT INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Instructor

Mr. Nguyễn Thành An

HO CHI MINH CITY, YEAR 2024

ACKNOWLEDGEMENT

We sincerely thank Mr. Nguyễn Thành An for teaching us the Introduction to Artificial Intelligence course with great enthusiasm. We want to express our deep appreciation for the dedication and professional knowledge that you shared with us. Through your classes, we gained a better understanding of the fundamental aspects of the Introduction to Artificial Intelligence, thanks to your detailed explanations and practical applications. You helped us grasp the knowledge and apply it effectively. Finally, we extend our heartfelt gratitude to Mr. Nguyễn Thành An for your commitment and invaluable support throughout our learning journey in this course. The skills and knowledge we acquired will continue to impact our future development. We sincerely thank you and wish your health, success, and happiness.

Ho Chi Minh City, May 19, 2024

Authors:

Nguyễn Nhật Chiêu

Nguyễn Đình Việt Hoàng

Đặng Công Minh

Trần Thiên Ân

Võ Minh Tài

DECLARATION OF AUTHORSHIP

Our group assures that this is our own report and was guided by Mr. Nguyễn Thành An. The research content and results in this report are honest and have not been published in any form before. The figures in the tables used for analysis, comments, and evaluations were collected by the authors from various sources clearly stated in the reference section.

Additionally, the report includes some comments, evaluations, and data from other authors and organizations, all of which are cited and noted for their origin.

If any fraud is detected, we fully take responsibility for the content of our midterm report for the second semester of the 2023-2024 academic year. Ton Duc Thang University is not involved in any copyright or intellectual property violations that we may cause during the process (if any).

Ho Chi Minh City, May 19, 2024

Authors:

Nguyễn Nhật Chiêu

Nguyễn Đình Việt Hoàng

Đặng Công Minh

Trần Thiên Ân

Võ Minh Tài

ABSTRACT

Chapter 2: 8x8 Tic-Tac-Toe

Introduction 8x8 Tic-Tac-Toe is a variant of the classic game played on a larger 8x8 grid, offering more strategic depth and complexity. The goal remains to be the first player to form a line of three symbols horizontally, vertically, or diagonally.

Advantages and Disadvantages

- **Advantages:**
  - Increased complexity and longer gameplay.
  - Diverse strategies due to more move possibilities.
- **Disadvantages:**
  - Longer playtime and potential for drawn-out endings.
  - Increased complexity can be overwhelming.
  - Higher likelihood of ties.

Approaches to Solve Tasks

1. **Initialization and Board Printing:**
   - Initialize the board and set the starting player.
   - Print the board with appropriate symbols.
2. **Game State Evaluation:**
   - Check for a winner by examining lines from each cell.
   - Check for a draw by ensuring no empty cells remain.

- Evaluate the board to assign scores based on potential winning patterns.

3. **Player Interaction:**
    - Get and validate player moves.
4. **Alpha-Beta Pruning:**
    - Implement the Alpha-Beta search algorithm to optimize AI moves.
5. **Main Game Loop:**
    - Run the game loop, alternating turns between the player and AI until a win or draw is detected.

Chapter 3: N-Queens with CNFs

Introduction The N-Queens problem involves placing N queens on an NxN chessboard such that no two queens threaten each other. CNF (Conjunctive Normal Form) is used to represent and solve this problem by encoding constraints into a standard logical format.

Advantages and Disadvantages

- **Advantages:**
    - Formal representation and leverage of SAT solvers.
    - Scalable to larger board sizes.
- **Disadvantages:**
    - Computational and encoding complexity.
    - Dependence on the performance of SAT solvers.

Real-Life Illustrative Example

- Represent the chessboard as a grid of variables.
- Encode constraints for row, column, and diagonal placements.
- Use a SAT solver to find a solution and display it.

Approaches to Solve Tasks

1. **Get Board Size:** Ensure the size (N) is valid.
2. **Create Variables:** Assign unique variables to each board cell.
3. **Generate Constraints:** Encode constraints to ensure one queen per row, column, and no diagonal conflicts.
4. **Solve N-Queens Problem:** Use a SAT solver to solve the constraints.
5. **Display Solution:** Visualize the board with placed queens.

Chapter 4: Decision Trees

Introduction Decision trees are used for classification and regression, representing decisions and their possible consequences in a tree-like model. They partition data based on feature values to create homogeneous subsets.

Advantages and Disadvantages

- **Advantages:**
  - Transparent and interpretable.
  - Can handle non-linear relationships and mixed data types.
- **Disadvantages:**
  - Prone to overfitting and sensitive to data changes.

- Bias towards features with more levels.

Real-Life Illustrative Examples

- Customer segmentation.
- Disease diagnosis.
- Financial investment decisions.
- Prioritizing work tasks.

Approaches to Solve Tasks

1. **Load Data:** Read data from a CSV file.
2. **Calculate Entropy:** Measure disorder within a data column.
3. **Calculate Average Entropy:** Evaluate the entropy of data subsets.
4. **Calculate Information Gain:** Determine the reduction in entropy when splitting data by an attribute.
5. **Evaluate Model:** Assess the decision tree's accuracy on test data.
6. **Visualize Decision Tree:** Create and save a graphical representation of the decision tree.
7. **Train Decision Tree:** Fit a decision tree classifier to the training data.
8. **Main Execution:** Integrate all functions to execute the complete workflow, including data loading, model training, evaluation, and visualization.

INSTRUCTOR RUBRIC

Supervisor’s Name: ……………………………………………………………..............

Comments: ……………………………………………………………………………...

Total Score Based on Rubric Evaluation: ………………………………………………

Ho Chi Minh City, date … month … year …

Supervisor

(sign and write your full name)

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1 – INTRODUCTION

1.1 Task assignment table

Table 1.1: Task assignment table

MEMBER	STUDENTID	MISSION	COMPLETE
NGUYỄN ĐÌNH VIỆT HOÀNG	522H0120	Task 3 - Task 4	100%
NGUYỄN NHẬT CHIÊU	522H0133	Task 2 - Task 3	100%
ĐẶNG CÔNG MINH	522H0095	Synthetic - Edit and proofread	100%
TRẦN THIÊN ÂN	522H0165	Task 1 - Task 2	100%
VÕ MINH TÀI	522H0168	Task 1 - Task 4	100%

1.2 Complete table

Table 1.2: Complete table

STT	Task	Finished
1	Task 1	100%
2	Task 2	100%
3	Task 3	100%
4	Task 4	100%

CHAPTER 2 – Task 1: 8X8 TIC-TAC-TOE

2.1 Brief introduction

8X8 Tic-Tac-Toe is a variant of the classic Tic-Tac-Toe game played on an 8x8 grid, which provides a larger playing area and more strategic possibilities. The objective of the game is still the same: to be the first player to form a line of three of their own symbols (traditionally "X" or "O") either horizontally, vertically, or diagonally.

In 8X8 Tic-Tac-Toe, the larger grid introduces additional complexity and challenges compared to the standard 3x3 version. With 64 cells to choose from, players have more options for placing their symbols, requiring them to think ahead and consider multiple lines of play simultaneously.

The gameplay mechanics are similar to regular Tic-Tac-Toe. Two players take turns placing their symbols on the board until one player achieves a winning line or the board is filled, resulting in a draw. The larger grid size adds an extra layer of strategy as players need to anticipate moves, block their opponent's potential winning lines, and create their own winning opportunities on a larger scale.

2.2 Advantages versus disadvantages

- Advantages:

+ Increased Complexity: The larger grid size of 8x8 adds complexity to the game compared to the traditional 3x3 version of Tic-Tac-Toe.

+ Longer Gameplay: With more spaces to fill, 8x8 Tic-Tac-Toe tends to have longer gameplay compared to the standard version.

+ Diverse Strategies: The larger grid allows for a wider range of strategies and move possibilities. Players have more opportunities to plan their moves, consider multiple winning patterns, and anticipate their opponents' moves.

- **Disadvantages**:

  + Longer Playtime: While longer gameplay can be an advantage for some players, it can also be a disadvantage for those seeking a quick and casual game.

  + Increased Complexity: The larger grid size can make the game more complex and potentially overwhelming for players who are new to Tic-Tac-Toe or prefer simpler games.

  + Drawn-out Endings: With more spaces to fill, 8x8 Tic-Tac-Toe games can often result in drawn-out endings, where both players struggle to secure a winning pattern.

  + Potential for More Ties: As the grid size increases, the likelihood of ending the game in a tie or draw also increases.

2.3 Approaches to solve tasks

I/ Initialization and Board Printing:

1. <u>Initialize Game Board and Set Starting Player:</u>

```
function InitializeGame():
   board ← Create BOARD_SIZE x BOARD_SIZE matrix filled with EMPTY
   current_player ← PLAYER_X
   return board, current_player
```

2. <u>Print the Board:</u>

```
function PrintBoard(board):
   print column indices
   for each row in board:
      print row index
```

```
        for each cell in row:
            print corresponding symbol (".", "X", "O") for EMPTY, PLAYER_X,
PLAYER_O respectively
```

II/ Game State Evaluation:

3. <u>Check for a Winner:</u>

```
function CheckWinner(board, player):
    for each cell in board:
        if horizontal line from cell contains WIN_LENGTH of player:
            return True
        if vertical line from cell contains WIN_LENGTH of player:
            return True
        if diagonal (both directions) line from cell contains WIN_LENGTH of player:
            return True
    return False
```

4. <u>Check for a Draw:</u>

```
function IsDraw(board):
    for each row in board:
        if EMPTY exists in row:
            return False
    return True
```

5. <u>Evaluate the Board:</u>

```
function EvaluateBoard(board):
   score ← 0
   for each horizontal, vertical, and diagonal line in board:
      score += EvaluateLine(line, PLAYER_X)
      score -= EvaluateLine(line, PLAYER_O)
   return score


function EvaluateLine(line, player):
   count ← count occurrences of player in line
   if count == WIN_LENGTH:
      return 10000
   elif count == WIN_LENGTH - 1 and one EMPTY:
      return 5000
   elif count == WIN_LENGTH - 2 and two EMPTY:
      return 1000
   return 0
```

III/ Player Interaction:

6. <u>Get Player Move:</u>

```
function GetPlayerMove(board):
    while True:
        try:
            move ← get input "Enter your move (row and column): "
            row, col ← parse move to integers
            if move is valid (within bounds and cell is EMPTY):
                return row, col
            else:
                print "Invalid move"
        except error:
            print "Invalid input"
```

IV/ Alpha-Beta Pruning:

7. <u>Alpha-Beta Search Algorithm:</u>

```
function AlphaBetaSearch(depth, alpha, beta, maximizing_player):
    if depth == 0 or terminal state (winner or draw):
        return EvaluateBoard(board), null


    if maximizing_player:
        return Maximize(depth, alpha, beta)
    else:
        return Minimize(depth, alpha, beta)

function Maximize(depth, alpha, beta):
    max_eval ← -∞
    best_move ← null
    for each move in sorted empty positions:
        apply move to board
        eval, _ ← AlphaBetaSearch(depth - 1, alpha, beta, False)
        undo move on board
        if eval > max_eval:
            max_eval ← eval
            best_move ← move
        alpha ← max(alpha, eval)
        if beta <= alpha:
            break
    return max_eval, best_move

function Minimize(depth, alpha, beta):
    min_eval ← ∞
    best_move ← null
    for each move in sorted empty positions:
        apply move to board
        eval, _ ← AlphaBetaSearch(depth - 1, alpha, beta, True)
        undo move on board
        if eval < min_eval:
            min_eval ← eval
            best_move ← move
        beta ← min(beta, eval)
        if beta <= alpha:
            break
    return min_eval, best_move


function SortMoves(moves):
    center ← BOARD_SIZE // 2
    return sorted moves based on distance to center
```

V/ Main Game Loop:

8. Main Function to Run the Game:

```
function Main():
    board, current_player ← InitializeGame()
    search_strategy ← Create SearchStrategy(board)

    while True:
        clear console
        PrintBoard(board)
        if current_player == PLAYER_X:
            row, col ← GetPlayerMove(board)
        else:
            _, move ← AlphaBetaSearch(DEPTH, -∞, ∞, False)
            row, col ← move
            print "AI chooses move: row col"

        board[row][col] = current_player

        if CheckWinner(board, current_player):
            clear console
            PrintBoard(board)
            print "Player X/O wins!"
            break

        if IsDraw(board):
            clear console
            PrintBoard(board)
            print "The game is a draw!"
            break

        current_player ← PLAYER_O if current_player == PLAYER_X else PLAYER_X
```

CHAPTER 3 – Task 2: N-QUEENS WITH CNFS

3.1 Brief introduction

The N-Queens problem is a classic puzzle that involves placing N queens on an NxN chessboard in such a way that no two queens threaten each other. The problem can be solved using various algorithms and techniques, one of which is the CNF (Conjunctive Normal Form) representation.

CNF is a standard format used in propositional logic, and it represents logical formulas as a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. In the context of the N-Queens problem, CNF can be used to encode the constraints and find a valid solution.

To solve the N-Queens problem using CNF, the chessboard is represented as a 2D grid, where each cell represents a variable. The variables can take two possible values: true (queen placed) or false (no queen). The constraints are then encoded as CNF clauses.

Using CNF for the N-Queens problem allows for leveraging existing tools and algorithms developed for solving SAT problems. It provides a systematic and formal approach to finding solutions and can be extended to larger board sizes or variations of the problem.

3.2 Advantages versus disadvantages

- Advantages:

+ Formal representation: CNF provides a formal and structured representation of the N-Queens problem, allowing for clear encoding of constraints and logical formulas.

+ Leveraging SAT solvers: CNFs enable the utilization of SAT solvers, which are powerful and efficient tools specifically designed to solve satisfiability problems.

+ Standardized format: CNF is accepted standard format in propositional logic, making it easier to communicate and share problem and solutions with others.

+ Scalability: The CNF representation allows for scalability to larger board sizes.

- Disadvantages:

+ Computational complexity: The N-Queens problem is known to be NP-complete, meaning that it becomes exponentially difficult as the board size increases.

+ Encoding complexity: While CNF encoding provides a systematic approach, it can be complex and time-consuming to properly encode all the necessary constraints for the N-Queens problem.

+ Limited problem scope: The CNF representation primarily focuses on solving the standard N-Queens problem, where the objective is to find a single valid solution.

+ Dependency on SAT solvers: The effectiveness of solving the N-Queens problem with CNFs heavily relies on the performance and capabilities of SAT solvers.

3.3 Real-life illustrative example

- Representation: We represent the 8x8 chessboard as a 2D grid, where each cell represents a variable. The variables can take the values true (queen placed) or false.

- Constraints encoding:

+ Exactly one queen per row: We create a CNF clause for each row, stating that at least one cell in that row must be true.

+ Exactly one queen per column: Similarly, we create a CNF clause for each column, ensuring that at least one cell in each column is true.

+ No 2 queens on the same diagonal: We encode the constraint that no 2 queens can be on the same diagonal by creating CNF clauses that check for diagonal conflicts.

- CNF formulation: We combine all the CNF clauses representing the constraints into a single CNF formula, which is the conjunction (AND) of all the clauses.

- SAT solving: We employ a SAT solver to find a satisfying assignment for the CNF formula. If a satisfying assignment is found, it represents a valid solution to the 8-Queens problem. The solver will assign true or false values to the variables, indicating the placement of queens on the chessboard.

3.4 Approaches to solve tasks

I/ Pseudocode for N-Queens Solver:

1. Get Board Size:

```
function GetBoardSize():
    while True:
        try:
            n = input("Enter the size of the board (N): ")
            if n >= 4:
                return n
            else:
                print("N must be at least 4.")
        except ValueError:
            print("Please enter a valid positive integer.")
```

2. <u>Create Variables:</u>

```
function CreateVariables(n):
    vars = {}
    counter = 1
    for row in range(0, n):
        for col in range(0, n):
            vars[(row, col)] = counter
            counter += 1
    return vars
```

3. <u>Generate Constraints:</u>

```
function GenerateConstraints(n, vars):
    constraints = []

    # One queen per row
    for row in range(0, n):
        row_constraints = []
        for col in range(0, n):
            row_constraints.append(vars[(row, col)])
        constraints.append(row_constraints)
        for col1 in range(0, n):
            for col2 in range(col1 + 1, n):
                constraints.append([-vars[(row, col1)], -vars[(row, col2)]])

    # One queen per column
    for col in range(0, n):
        col_constraints = []
        for row in range(0, n):
            col_constraints.append(vars[(row, col)])
        constraints.append(col_constraints)
        for row1 in range(0, n):
            for row2 in range(row1 + 1, n):
                constraints.append([-vars[(row1, col)], -vars[(row2, col)]])

    # No two queens on the same diagonal
    for row in range(0, n):
        for col in range(0, n):
            for step in range(1, n):
                if row + step < n and col + step < n:
                    constraints.append([-vars[(row, col)], -vars[(row + step, col + step)]])
                if row + step < n and col - step >= 0:
                    constraints.append([-vars[(row, col)], -vars[(row + step, col - step)]])
                if row - step >= 0 and col + step < n:
                    constraints.append([-vars[(row, col)], -vars[(row - step, col + step)]])
                if row - step >= 0 and col - step >= 0:
                    constraints.append([-vars[(row, col)], -vars[(row - step, col - step)]])

    return constraints
```

4. <u>Solve N-Queens Problem:</u>

```
function SolveNQueens(constraints):
    solver = Create Glucose3 solver with constraints
    if solver.solve():
        return solver.get_model()
    else:
        return None
```

5. <u>Display Solution:</u>

```
function DisplaySolution(n, solution, vars):
    if solution:
        board = Create n x n board filled with '.'
        for var in solution:
            if var > 0:
                (row, col) = Find key in vars where value == var
                board[row][col] = 'Q'
        for row in board:
            print row
    else:
        print "No solution exists"
```

6. Main Execution:

```
function Main():
    n = GetBoardSize()
    vars = CreateVariables(n)
    constraints = GenerateConstraints(n, vars)
    solution = SolveNQueens(constraints)
    DisplaySolution(n, solution, vars)


if __name__ == "__main__":
    Main()
```

II/ Explanation:

1. GetBoardSize: Prompts the user to input the size of the board (N) and ensures it's at least 4.
2. CreateVariables: Creates a unique variable for each cell on the board.
3. GenerateConstraints: Generates the CNF constraints for the N-Queens problem.
    - Ensures one queen per row.
    - Ensures one queen per column.
    - Ensures no two queens share the same diagonal.
4. SolveNQueens: Uses the Glucose3 solver to solve the problem based on the generated constraints.
5. DisplaySolution: Displays the board with queens placed based on the solution.
6. Main: Combines all functions to execute the N-Queens solver.

CHAPTER 4 – Task 3: DECISION TREES

4.1 Brief introduction

Decision trees are a popular machine learning algorithm used for both classification and regression tasks. They are a flowchart-like structure where internal nodes represent features or attributes, branches represent decisions or rules, and leaf nodes represent the outcome or prediction.

The main idea behind decision trees is to partition the data based on the values of different features, aiming to create homogeneous subsets of data with similar target values. This partitioning is done recursively, leading to a tree-like structure where each internal node represents a decision and each leaf node represents a class label or a predicted value.

The construction of a decision tree involves selecting the best features to split the data at each internal node based on certain criteria such as information gain, Gini impurity, or entropy. These criteria measure the impurity or disorder of the data, and the goal is to minimize impurity and maximize information gain with each split.

Once a decision tree is constructed, it can be used to make predictions on new, unseen data by traversing the tree from the root node to a leaf node based on the feature values of the instance being classified. The predicted class label or value associated with the reached leaf node is then assigned as the final prediction.

4.2 Advantages versus disadvantages

- Advantages:

+ Interpretability: Decision trees provide a transparent and interpretable model since the decision-making process is represented in a tree-like structure.

+ Feature importance: Decision trees can measure the importance of features in the prediction process.

+ Handling non-linear relationships: Decision trees can capture non-linear relationships and interactions between features, making them suitable for complex datasets.

+ Handling mixed data types: Decision trees can handle both categorical and numerical features without requiring explicit feature encoding or normalization.

- **Disadvantages:**

+ Overfitting: Decision trees are prone to overfitting, meaning they can create overly complex models that perform well on the training data but generalize poorly to unseen data.

+ Lack of robustness: Decision trees are sensitive to small changes in the data, which can lead to different tree structures and predictions.

+ Bias towards features with more levels: Decision trees tend to favor features with more levels or categories since they can potentially provide more splits and information gain.

4.3 Real-life illustrative example

- Customer segmentation: A company can use a decision tree to classify customers into different groups based on features such as age, income, product preferences, and past buying behavior. The decision tree helps the company identify potential customer segments, enabling them to create targeted marketing strategies and advertisements to increase sales and optimize marketing campaigns.

- Disease diagnosis: In the field of healthcare, decision trees can be used to support disease diagnosis. Based on a patient's symptoms and test results, a decision tree can determine potential diseases or disease clusters that may cause similar symptoms. This helps doctors or healthcare systems make accurate and prompt diagnostic decisions.

- Financial investment decisions: Decision trees can be applied in the financial sector to make investment decisions. Based on financial indicators, market information, and economic factors, decision trees can predict whether a company or an industry has growth potential. This helps investors make informed decisions about investing in stocks, funds, or other financial assets.

- Prioritizing work tasks: In project management, decision trees can be used to prioritize work tasks. Based on factors such as importance, feasibility, and time required, a decision tree can determine which tasks should be done and which tasks can wait. This helps increase efficiency and ensures that critical tasks are completed on time.

4.4 Approaches to solve tasks

I/ Pseudocode for Decision Tree Classification and Evaluation:

1. <u>Load Data:</u>

```
function LoadData(file_path):
    return ReadCSV(file_path)
```

2. <u>Calculate Entropy:</u>

```
function CalculateEntropy(column):
    value_counts = GetValueCounts(column)
    probabilities = value_counts / Length(column)
    entropy = -sum(probabilities * log2(probabilities))
    return entropy
```

3. <u>Calculate Average Entropy:</u>

```
function CalculateAverageEntropy(data, attribute):
    attribute_values = GetUniqueValues(data[attribute])
    average_entropy = 0

    for value in attribute_values:
        subset = FilterDataByAttributeValue(data, attribute, value)
        subset_entropy = CalculateEntropy(subset['Rank'])
        average_entropy += (Length(subset) / Length(data)) * subset_entropy

    return average_entropy
```

4. <u>Calculate Information Gain:</u>

```
function CalculateInformationGain(data, attribute):
    entropy_before = CalculateEntropy(data['Rank'])
    average_entropy = CalculateAverageEntropy(data, attribute)
    information_gain = entropy_before - average_entropy
    return information_gain
```

5. <u>Evaluate Model:</u>

```
function EvaluateModel(model, X_test, y_test):
    y_pred = model.Predict(X_test)
    accuracy = CalculateAccuracy(y_test, y_pred)
    Print "Decision Tree Accuracy: ", accuracy
    return accuracy
```

6. <u>Visualize Decision Tree:</u>

```
function VisualizeDecisionTree(model, feature_names, class_names, file_name):
    dot_data = CreateStringIO()
    ExportGraphviz(model, out_file=dot_data,
                   filled=True, rounded=True,
                   special_characters=True, feature_names=feature_names,
                   class_names=class_names)
    graph = CreateGraphFromDotData(dot_data)
    graph.WritePNG(file_name)
    return DisplayImage(graph)
```

7. <u>Train Decision Tree:</u>

```
function TrainDecisionTree(X_train, y_train, random_state=42):
    dt_classifier = CreateDecisionTreeClassifier(random_state)
    dt_classifier.Fit(X_train, y_train)
    return dt_classifier
```

8. <u>Main Execution:</u>

```
function Main():
    # Load data
    data = LoadData("file_path.csv")

    # Prompt user for attribute
    attribute = Input("Enter the name of a score attribute (Q1 to Q9): ")

    # Calculate and print entropy
    entropy = CalculateEntropy(data['Rank'])
    Print "Entropy (H): ", entropy

    # Calculate and print average entropy
    average_entropy = CalculateAverageEntropy(data, attribute)
    Print "Average Entropy (AE) for ", attribute, ": ", average_entropy

    # Calculate and print information gain
    information_gain = CalculateInformationGain(data, attribute)
    Print "Information Gain (IG) for ", attribute, ": ", information_gain

    # Split data into features and target
    X = SelectColumns(data, from=2, to=End)
    y = data['Rank']
    X_train, X_test, y_train, y_test = TrainTestSplit(X, y, test_size=0.2,
random_state=42)

    # Train decision tree
    dt_classifier = TrainDecisionTree(X_train, y_train)

    # Evaluate model
    EvaluateModel(dt_classifier, X_test, y_test)

    # Visualize decision tree
    VisualizeDecisionTree(dt_classifier, GetColumnNames(X), GetUniqueValues(y),
"decision_tree.png")

if __name__ == "__main__":
    Main()
```

II/ Explanation:

1. LoadData: Loads data from a CSV file into a DataFrame.
2. CalculateEntropy: Calculates the entropy of a given column.
3. CalculateAverageEntropy: Computes the average entropy for subsets of data based on a specified attribute.
4. CalculateInformationGain: Calculates the information gain of an attribute by comparing the entropy before and after splitting on the attribute.
5. EvaluateModel: Evaluates the decision tree model's accuracy on test data.
6. VisualizeDecisionTree: Generates and saves a visual representation of the decision tree.
7. TrainDecisionTree: Trains a decision tree classifier on the training data.
8. Main: Executes the main workflow: loading data, calculating entropies and information gain, splitting data, training and evaluating the model, and visualizing the decision tree.

REFERENCES

1. I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, "Data Mining: Practical Machine Learning Tools and Techniques," 4th ed., Morgan Kaufmann, 2016.
2. J. R. Quinlan, "C4.5: Programs for Machine Learning," Morgan Kaufmann, 1993.
3. L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees," Wadsworth, 1984.
4. S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 3rd ed., Prentice Hall, 2009.
5. T. M. Mitchell, "Machine Learning," McGraw-Hill, 1997.
6. R. E. Schapire and Y. Freund, "Boosting: Foundations and Algorithms," MIT Press, 2012.
7. R. Agrawal, T. Imieliński, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," in *Proc. 1993 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '93),* Washington, D.C., 1993, pp. 207-216.
8. C. Bishop, "Pattern Recognition and Machine Learning," Springer, 2006.
9. R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, Eds. New York: Plenum, 1972, pp. 85-103.
10. M. Garey and D. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, 1979.