ELECTRICAL ENGINEERING
COMPUTER SYSTEMS ENGINEERING
CCE414 & ELE355– COURSE PROJECT
(TERM 2)

Project 1 & 2
Course: CCE414 & ELE355 (Artificial Intelligence)

S#	Student Name	Student ID
1	Malak Mounir Abdellatif	231903643
2	Razan Ahmed Fawzi	221903165
3	Farida Waheed Abdel Bary	221903168
4	Nour Hesham Elsayed	221902960
5	Nourhan Farag Mohamed	231903707
6		

Marks	
Free Topic Description	**/40**

Problem solving as a search problem	Problem formulation	**/40**
	Implementation/Evaluation /Sample output	**/60**
	Project interface & Poster Design	**/25**
	Report Style and Formatting	**/15**
	Presentation / response to questions	**/20**
	Creativity (**Bonus**)	**/10**

Expert System in Prolog	Implementation/Evaluation /Sample output	**/60**
	Project interface & Poster Design	**/40**

Total	**/300**

CCE414 Artifcial Intelligence

CONNECT 4 AI PROJECT 1 REPORT

Presented to:

DR. Mohamed Rehan

Due date: 3 / 5 / 2025

Date Handed in: 3 / 5 / 2025

GROUP 1

Malak Mounir Abdellatif	231903643
Razan Ahmed Fawzi	221903165
Farida Waheed Abdelbary	221903168
Nour Hesham Elsayed	221902960
Nourhan Farag Mohamed	231903707

TABLE OF CONTENTS

01 Problem Formulation

02 Technical Discussion

03 Results Discussion

04 Results

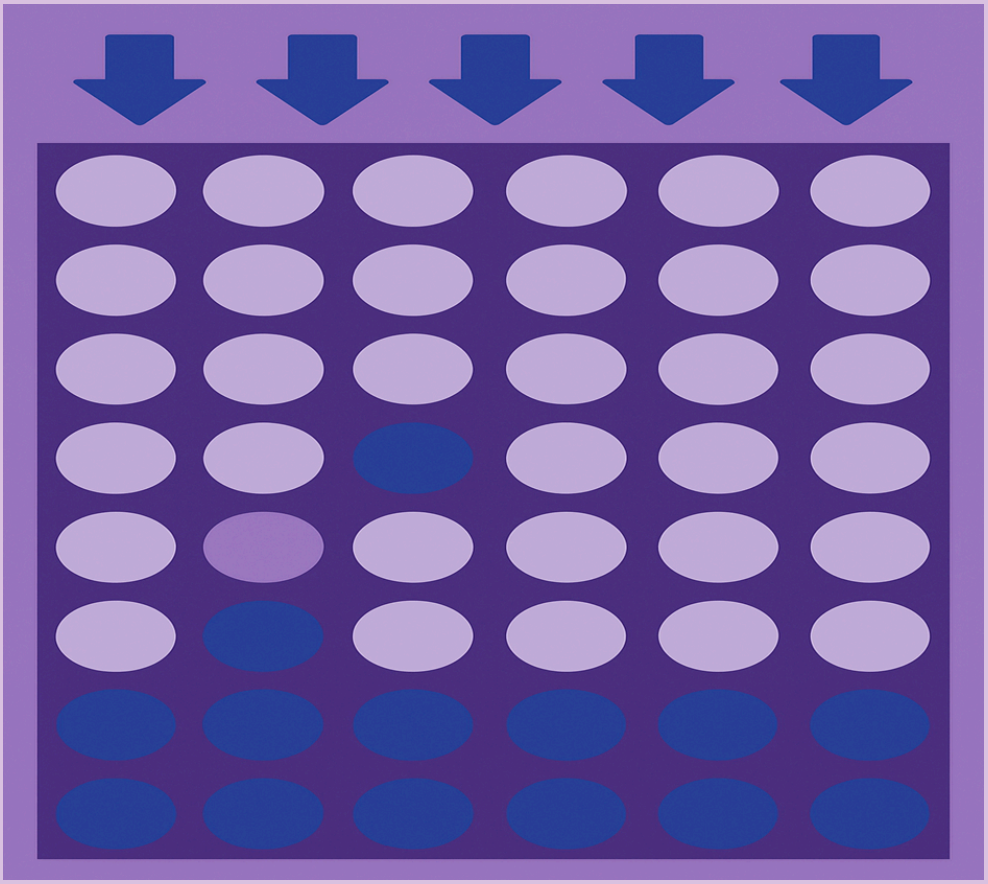
05 Free Topic

06 Task Assignment

INTRODUCTION

This project explores the implementation of artificial intelligence algorithms in the context of the popular board game Connect 4. The objective is to develop an AI system capable of competing with human players or other AI systems using classical search strategies. Connect 4 is a well-known, turn-based game that involves strategic placement of
discs on a vertical grid. The goal is to be the first to connect four of one's own discs in a row—horizontally, vertically, or diagonally.

The motivation behind this project is to apply AI search algorithms, such as Minimax, Alpha-Beta Pruning, and Iterative Deepening, to make intelligent decisions in a competitive game environment. A graphical user interface (GUI) was also developed to enable interactive play, visualization of AI thinking processes, and real-time comparison of different algorithm strategies.



PROBLEM FORMULATION

Initial State:

The game starts with an empty **6×7 grid** (6 rows, 7 columns).
Two players take turns: one plays with **red** discs and the other with **yellow** discs. The board is initially empty, and Red typically plays first.

Actions:

Legal Moves: A player selects one of the **7 columns** to drop a disc into. The disc **falls to the lowest available row** in the chosen column.

Invalid Moves:

- Choosing a column that is already full (i.e., has 6 discs) is illegal.
- Discs cannot be inserted in the middle or on top of other discs unless supported from below.

Transition Model:

Given a current state and an action (chosen column), the transition model:

- Places the current player's disc in the lowest available row of the selected column.
- Updates the game board.
- Switches the turn to the next player.

Goal Test:

A state is a goal state if the most recent move results in:

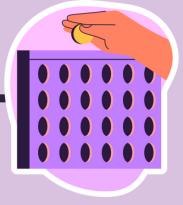
- Four of the same color discs aligned:
  - Horizontally
  - Vertically
  - Diagonally (either direction)

If this happens, the player who made the move wins.

Path Cost:

Each move has a uniform cost of one unit.
The path cost can be measured as:

- The number of moves made from the initial state to reach a terminal state.
- Optionally, in AI planning, you could define separate evaluation metrics (e.g., minimizing the opponent's chances).



TECHNICAL DISCUSSION

Minimax Algorithm

The Minimax algorithm serves as the foundational decision-making framework for our Connect 4 AI implementation. This adversarial search algorithm operates on the principle of optimal play, systematically evaluating all possible future game states to determine the most advantageous move while assuming the opponent will respond optimally.

Algorithmic Principle

The algorithm employs a recursive depth-first search approach that:

1. Alternates between maximizing (AI's turn) and minimizing (opponent's turn) perspectives
2. Explores the game tree to a specified depth limit
3. Returns an evaluation of the current board position

Complete	Yes (finite trees)
Optimal	Yes (vs optimal opponent)
Time	$O(b^m) \rightarrow \sim 7^d$ in Connect 4
Space	O(m) → Depth-first stack

Recursive Evaluation Process

At each recursive step:

1. **Maximizing Player (AI Turn):**
   - Evaluates all legal moves
   - Selects the move yielding the highest score (optimal play for the AI)
2. **Minimizing Player (Opponent Turn):**
   - Evaluates all legal responses
   - Selects the move yielding the lowest score (optimal counterplay against the AI)
3. **Terminal State Conditions:**
   - AI victory: Returns +∞ (or sufficiently large positive value)
   - Opponent victory: Returns -∞ (or sufficiently large negative value)
   - Draw condition: Returns 0
   - Depth limit reached: Returns heuristic evaluation of the position

Mathematical Formulation

The recursive scoring function can be expressed as:

$$\text{minimax}(s, d) = \begin{cases} \text{utility}(s) & \text{if } s \text{ is terminal or } d = 0 \\ \max_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a), d - 1) & \text{if player} = \text{AI} \\ \min_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a), d - 1) & \text{if player} = \text{opponent} \end{cases}$$

where:
s = current game state
d = remaining search depth
a = possible action/move
result(s,a) = new state after applying action ***a*** to state ***s***

TECHNICAL DISCUSSION

How Minimax Calculates Scores?

Example

```
def evaluate_window(self, window, piece, opponent_piece):
    score = 0
# Prioritize blocking opponent wins
    if window.count(opponent_piece) == 3 and window.count(0) == 1:
        return -100  # Urgent block
# Reward potential wins
    if window.count(piece) == 3 and window.count(0) == 1:
        score += 50   # Immediate win chance
    elif window.count(piece) == 2 and window.count(0) == 2:
        score += 10   # Potential future win
    return score
```

Heuristic Breakdown

- **Winning Lines:**
  - AI_3_in_a_row * 100 (prioritize immediate wins/blocks).
  - AI_2_in_a_row * 10 (secondary threat building).
  - Player_3_in_a_row * (-150) (urgent opponent threat mitigation).

$$\text{Score} = (100 \times A_3) + (10 \times A_2) - (150 \times P_3) + \text{Center\_Bias}$$

- **Strategic Bias:**
  - Center_column_bias = +2 per AI disc (central control improves mobility).

```
def minimax(self, node, depth, maximizing_player, player_piece):
    valid_moves = self.get_valid_moves(node.board)
    terminal = node.is_terminal()

    if depth == 0 or terminal:
        if terminal:
            if node.check_win(node.board, player_piece):
                return None, math.inf
            elif node.check_win(node.board, 3 - player_piece):
                return None, -math.inf
            else:
                return None, 0
        return None, self.evaluate_position(node.board, player_piece)

    if maximizing_player:
        value = -math.inf
        best_move = valid_moves[0]
        for move in valid_moves:
            new_board = self.drop_disc(node.board, move, player_piece)
            child = Node(node, new_board, depth-1, 3 - player_piece, player_piece, move)
            _, new_score = self.minimax(child, depth-1, False, player_piece)
            if new_score > value:
                value = new_score
                best_move = move
        return best_move, value
    else:
        value = math.inf
        best_move = valid_moves[0]
        opponent_piece = 3 - player_piece
        for move in valid_moves:
            new_board = self.drop_disc(node.board, move, opponent_piece)
            child = Node(node, new_board, depth-1, 3 - opponent_piece, opponent_piece, move)
            _, new_score = self.minimax(child, depth-1, True, player_piece)
            if new_score < value:
                value = new_score
                best_move = move
        return best_move, value
```

TECHNICAL DISCUSSION

Alpha-Beta Pruning Algorithm

Alpha-Beta Pruning is an optimization of the Minimax algorithm that reduces computation time by eliminating branches that cannot influence the final decision. It maintains Minimax's optimality while significantly improving efficiency.

Algorithmic Principle

The algorithm enhances Minimax by:

- Tracking alpha (best already explored option for Max)
- Tracking beta (best already explored option for Min)
- Pruning branches when $\alpha \geq \beta$ (proven worse than current best)

Completeness	Yes (for finite trees)
Optimality	Yes (identical to Minimax)
Time	O(b^(d/2)) → ~$\sqrt{b^d}$ nodes
Space	O(d) → Depth-first stack

Recursive Evaluation Process

At each recursive step:

Maximizing Player (AI Turn):

- Updates alpha with the maximum value found
- Prunes when value ≥ beta
- Returns when alpha ≥ beta

Minimizing Player (Opponent Turn):

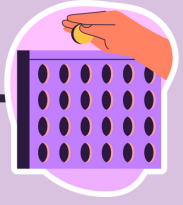
- Updates beta with minimum value found
- Prunes when value ≤ alpha
- Returns when beta ≤ alpha

Terminal Conditions for Alpha-Beta Pruning:

- Win/Loss/Draw Detection
  - Returns +∞ (AI wins), -∞ (opponent wins), or 0 (draw)
- Depth Limit Reached
  - Uses heuristic evaluation of board position
- Pruning Triggers
  - Stops evaluating a branch when:
    - Maximizer finds value ≥ β
    - Minimizer finds value ≤ α
  - Maintains Minimax's optimality while skipping irrelevant branches.

Why the Same Heuristic Works Better

- Faster deep searches: Evaluates same positions but 2-10x faster
- Earlier threat detection: Prunes branches where opponent's best reply is worse than current α/β
- Preserved optimality: Guaranteed identical results to Minimax, just more efficient



TECHNICAL DISCUSSION

```python
def alphabeta(self, node, depth, alpha, beta, maximizing_player, player_piece):
    valid_moves = self.get_valid_moves(node.board)
    terminal = node.is_terminal()

    if depth == 0 or terminal:
        if terminal:
            if node.check_win(node.board, player_piece):
                return None, math.inf
            elif node.check_win(node.board, 3 - player_piece):
                return None, -math.inf
            else:
                return None, 0
        return None, self.evaluate_position(node.board, player_piece)

    if maximizing_player:
        value = -math.inf
        best_move = valid_moves[0]
        for move in valid_moves:
            new_board = self.drop_disc(node.board, move, player_piece)
            child = Node(node, new_board, depth-1, 3 - player_piece, player_piece, move)
            _, new_score = self.alphabeta(child, depth-1, alpha, beta, False, player_piece)
            if new_score > value:
                value = new_score
                best_move = move
                alpha = max(alpha, value)
            if value >= beta:
                break
        return best_move, value
    else:
        value = math.inf
        best_move = valid_moves[0]
        opponent_piece = 3 - player_piece
        for move in valid_moves:
            new_board = self.drop_disc(node.board, move, opponent_piece)
            child = Node(node, new_board, depth-1, 3 - opponent_piece, opponent_piece, move)
            _, new_score = self.alphabeta(child, depth-1, alpha, beta, True, player_piece)
            if new_score < value:
                value = new_score
                best_move = move
                beta = min(beta, value)
            if value <= alpha:
                break
        return best_move, value
```

TECHNICAL DISCUSSION

Iterative Deepening (ID) with Alpha-Beta

An optimized search algorithm combining depth flexibility with pruning efficiency

Completeness	Yes (for finite trees, with time or depth limit)
Optimality	Yes (if full depth is reached)
Time	O(b^d) → multiple re-expansions due to deepening
Space	O(d) → depth-first search stack only

Algorithmic Principle

Iterative Deepening enhances Alpha-Beta by:

1. **Progressive Depth Search:**
   - Executes Alpha-Beta from depth=1 to max_depth (default=10)
   - Each iteration provides better move ordering for the next
2. **Time Management:**
   - Continues searching if time_limit=None (unlimited search)
   - Returns the best move found at the deepest completed depth
3. **Synergy with Alpha-Beta:**
   - Previous iterations improve move ordering for better pruning
   - Retains all benefits of Alpha-Beta's branch elimination

At each depth level:

Start:

Set alpha to -∞, beta to +∞, and prepare to track the best move.

Search:

Use iterative deepening—search one level deeper each time.

Stop if:

- Time runs out → return the current best move.
- Win/Loss is guaranteed (±∞) → exit early.
- Depth finished → store good moves to search smarter next time.

```python
def iterative_deepening_alphabeta(self, root, max_depth=10, time_limit=None):
    start_time = time.time()
    best_move = self.get_valid_moves(root.board)[0]
    best_value = -math.inf

    for depth in range(1, max_depth + 1):
      # Skip time check if time_limit is None
      if time_limit is not None and time.time() - start_time > time_limit * 0.8:
        break

      current_move, current_value = self.alphabeta(root, depth, -math.inf, math.inf, True, root.current_player)

    if current_move is not None and current_value > best_value:
        best_move = current_move
        best_value = current_value
        if best_value == math.inf:  # Early win
            return best_move

    return best_move
```

RESULTS DISCUSSION

The GUI enables human vs. AI and AI vs. AI modes. The AI uses the chosen algorithm to decide moves in real-time.
The visual interface includes game statistics, animated moves, and algorithm descriptions. The user can observe AI behavior and compare strategies.

MiniMax Algorithm

The MiniMax algorithm guarantees optimal moves by evaluating all possible game states. While it ensured the AI always played the best possible move, it suffered from high time complexity due to the need to explore the entire search tree. This made it impractical for real-time use, but it ensured the AI never lost and always either won or drew.

Alpha-Beta Pruning

Alpha-Beta Pruning optimizes the MiniMax algorithm by pruning branches of the game tree that do not affect the outcome, significantly improving efficiency without sacrificing optimality. This made the AI faster, while still guaranteeing the best move, making it more suitable for real-time play. It allowed the AI to make optimal moves faster than MiniMax, but the outcome was the same: the AI was unbeatable.

Iterative Deepening Alpha-Beta

Iterative Deepening Alpha-Beta combines the benefits of Alpha-Beta Pruning with Iterative Deepening, which adjusts the search depth based on available time. This made it ideal for time-constrained environments. The algorithm still made optimal moves, but when the time was limited, it sometimes resulted in a draw if the depth was not sufficient to find a winning move. This method provided a more flexible approach, balancing efficiency and optimality, especially in real-time scenarios.

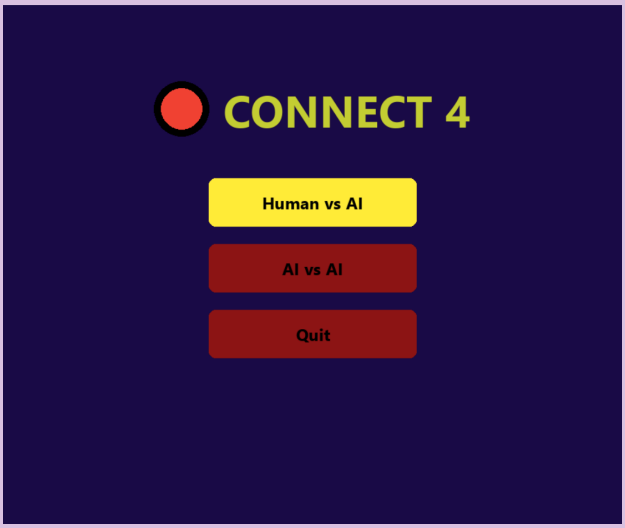
Comparison

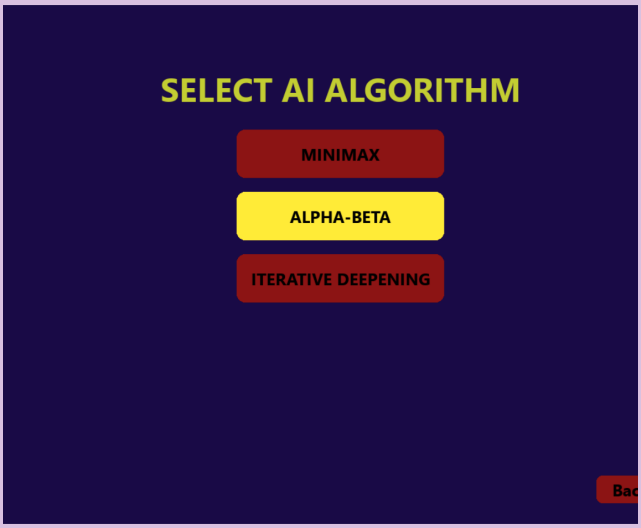
- ***Effectiveness:*** All algorithms ensured the AI did not lose, with MiniMax and Alpha-Beta guaranteeing a win or draw. Iterative Deepening Alpha-Beta, while occasionally drawing due to time limitations, performed similarly.
- ***Efficiency:*** MiniMax had the highest computational cost, while Alpha-Beta Pruning significantly reduced the time needed to find the best move. Iterative Deepening Alpha-Beta was the most flexible, adapting based on time constraints.
- ***Practical Use:*** For real-time play, Iterative Deepening Alpha-Beta was the best option, balancing speed and optimality. Alpha-Beta Pruning also performed well, while MiniMax was less practical due to slower decision-making.


YOU
WIN!

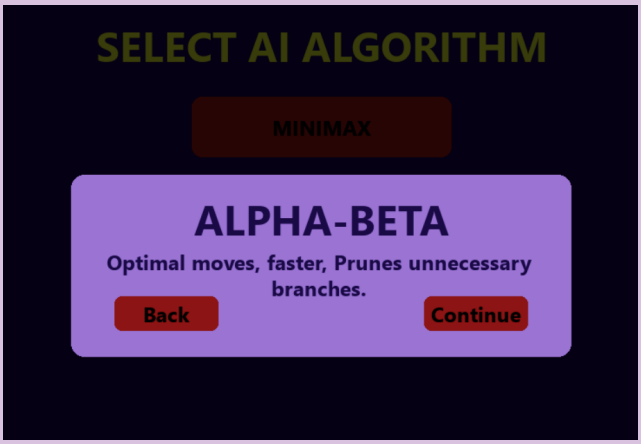
RESULTS

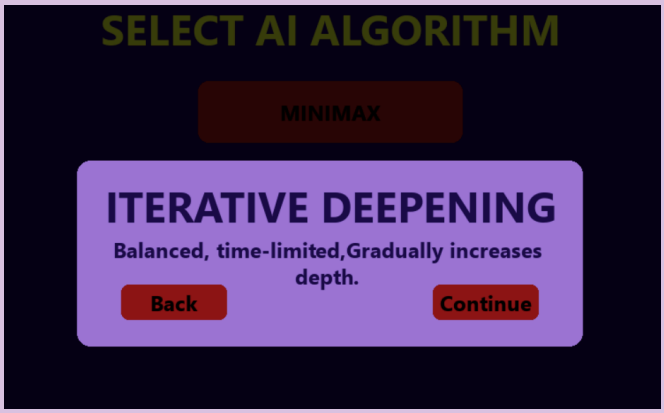

LOCAL
VISIT
90
VS
TIME
85

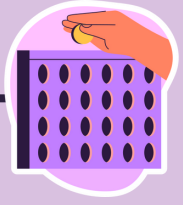

CONNECT 4
Human vs AI
AI vs AI
Quit


SELECT AI ALGORITHM
MINIMAX
ALPHA-BETA
ITERATIVE DEEPENING

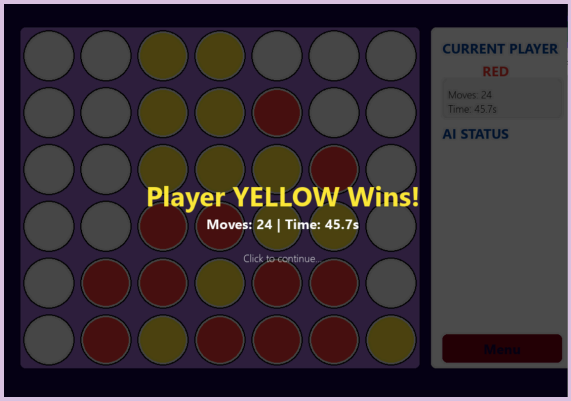

SELECT AI ALGORITHM
MINIMAX
MINIMAX
Perfect play but slower,Uses full depth search.
Back
Continue


SELECT AI ALGORITHM
MINIMAX
ALPHA-BETA
Optimal moves, faster, Prunes unnecessary branches.
Back
Continue


SELECT AI ALGORITHM
MINIMAX
ITERATIVE DEEPENING
Balanced, time-limited,Gradually increases depth.
Back
Continue

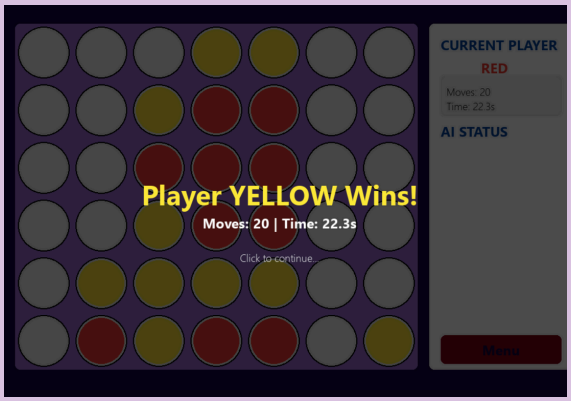


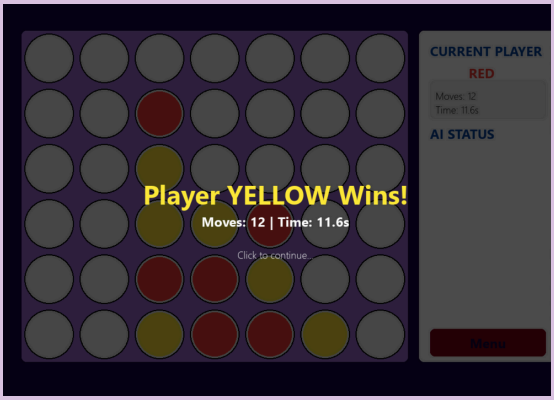
RESULTS



MINIMAX

ALPHA-BETA





ID ALPHA-BETA

AI VS AI

TASK ASSIGNMENT

Name	Task	%	Signature
Malak Mounir	Project 1 Connect Four implementation code. Expert system implementation. Project 1 Poster.	20%	
Farida Waheed	Project 1 Connect Four implementation code. Expert system implementation. Project 2 Poster	20%	
Nourhan Farag	Project 1 Connect Four implementation code. Expert system implementation. Project 1 Poster.	20%	
Razan Ahmed	Expert system implementation and report. Free topic example. Presentation.	20%	
Nour Hesham	Expert system implementation and report. Free topic example. Presentation.	20%