Khulna University of Engineering & Technology, Khulna

Dept. of Computer Science and Engineering



Course No.: CSE 4110

Course Title: Artificial Intelligence Laboratory

Project Report

**Pentago**

Submission Date:

| Submitted To | Submitted By |
| --- | --- |
| Most. Kaniz Fatema Isha Lecturer  Dept. of CSE, KUET, Khulna    Md. Shahidul Salim  Lecturer  Dept. of CSE, KUET, Khulna | Mahdi Bhuiyan  Roll: 1907015  Mahjabin Sultana  Roll: 1907020  Year: 4th  Term: 1st  Dept. of CSE, KUET, Khulna |

## 

## **Objectives**

1. To build an AI player for the game Pentago.

2. To implement the Minimax algorithm for AI move.

3. To optimize Minimax with Alpha-Beta Pruning.

4. To combine Iterative Deepening and Minimax for further optimization.

5. To implement Genetic Algorithm for AI move.

## **Introduction**

Pentago is one of the popular board games. This game requires two players. This is an abstract strategy game. A strong AI player can be developed for this game using different artificial intelligence techniques. The goal of this project is to build such an AI player. This project combines implementation of Minimax algorithm, Alpha-Beta pruning, Genetic Algorithm and Iterative Deepening to build the AI player. We have used Pygame to build the user interface.

## **Game Rules**

**1. Board Structure :** The game board is divided into 4 quadrants. Each of the quadrants has one 3x3 grid. The quadrants are rotatable in both clockwise and anticlockwise direction.

**2. Marbles :** Each player has 18 marbles. The AI has White marbles and the User has Black marbles.

**3. Moves :** Their move consists of placing a marble in any cell and then rotating one of the quadrants. After placing a marble, the user must rotate one quadrant otherwise the game will not progress. Users will not be able to place more than one marble.

**4. Winning Condition :** The first player to place five marbles in a row wins the game. The row can be horizontal, vertical or diagonal. After placing the fifth marble, there is no need to rotate any quadrant.

## 

## **Features**

The game starts with a welcome window. This window has two buttons. Play button takes the user to a window where he/she can select game mode. The game has 3 modes:

1. Easy

2. Medium

3. Hard

Game rules button shows the user the rules of the game. Users can go back to the previous window by clicking on the arrow on the right upper side. Users can place a marble by clicking on any of the 36 cells. Rotation can be done using the clockwise, anti-clockwise rotate symbol beside each quadrant. If AI or the user wins, a new window announces the winner.

## **User Interface**

| Figure 1: Welcome Screen | Figure 2: Users can select mode |
| --- | --- |
| Figure 3: Shows users the game rules | Figure 4: The game board |
| Figure 5: Shows if AI won | Figure 6: Shows if user won |

## **Tools Used**

| Editor | Visual Studio Code |
| --- | --- |
| Language | Python |
| Libraries | 1. NumPy 2. PyGame 3. Random 4. Copy |

## 

## **Implementation**

1. **The board structure and board movements are implemented in the “board.py” file. The pseudocode for this file is:**

Class Board:

Initialize:

Initialize board as an empty list

Initialize selected\_piece as None

Call create\_board()

Method create\_board:

For each row in the range ROWS:

Append an empty list to the board

For each column in the range COLS:

Append 0 to the current row

Method rotate\_board(board):

Return the board rotated 90 degrees clockwise

Method draw\_cubes(win):

For each row in the range ROWS:

Draw red and dark red squares alternately

For each row in the range ROWS:

For each column in the range COLS:

If board position is 1 (white):

Draw white circle at (x, y)

If board position is -1 (black):

Draw black circle at (x, y)

Method draw\_rotatesym(win):

Draw rotation symbols at designated positions on the board

Method construct\_board:

Return a 6x6 matrix filled with zeros

Method is\_board\_full(board):

For each line in the board:

For each value in the line:

If value is 0:

Return False

Return True

Method get\_position\_if\_valid(board, x, y):

If x or y is outside the range or position is already filled:

Return None

Return (x, y)

Method draw(selected\_row, selected\_col, turn):

Get the piece at the selected position

If piece is 0:

If turn is WHITE:

Set board position to 1

If turn is BLACK:

Set board position to -1

Method rotate\_board(start\_row, end\_row, start\_col, end\_col, rotate):

Convert board to numpy array

Extract portion of the board to rotate

If rotate is anticlockwise:

Rotate portion 90 degrees anticlockwise

If rotate is clockwise:

Rotate portion 270 degrees clockwise

Replace portion in original matrix

Convert matrix back to list

Method rotate(grid\_no, rotation):

Set start\_row, end\_row, start\_col, end\_col based on grid\_no

Call rotate\_board with calculated indices and rotation

Method get\_valid\_moves:

Initialize moves as an empty set

For each row in the range ROWS:

For each column in the range COLS:

If board position is 0:

Add all possible moves (row, col, grid, rotation) to moves

Return moves

Method winner:

Check for horizontal, vertical, main diagonal and anti-diagonal sequences of 1s

If found, return 1

Check for horizontal, vertical, main diagonal and anti-diagonal sequences of -1s

If found, return -1

Return 0 if no winner is found

1. **The game is controlled (like: if it is AI’s turn or user’s turn, update the board after each move) with the “game.py” file. The pseudocode for this file is:**

Class Game:

Method \_\_init\_\_(win, board):

Set self.board to board

Set self.turn to BLACK

Set self.valid\_moves to an empty dictionary

Set self.win to win

Set self.move to False

Set self.rotate to False

Method update():

Call self.board.draw\_cubes(self.win)

Call pygame.display.update()

Method change\_turn():

Set self.move to False

Set self.rotate to False

If self.turn is BLACK:

Set the window caption to 'Your Turn'

Set self.turn to WHITE

Else:

Set the window caption to 'AI Turn'

Set self.turn to BLACK

Method get\_board():

Return self.board

Method ai\_move(board, mode):

Set the window caption to 'AI Turn'

Set self.board to board

Call self.update()

Call self.change\_turn()

Method place\_marble(win, row, col):

If self.move is False and self.rotate is False:

If row and col are within valid range:

Call self.board.draw(row, col, self.turn)

Call pygame.display.update()

Set self.move to True

If self.turn is BLACK:

Decrement self.board.b\_left by 1

Else if self.turn is WHITE:

Decrement self.board.w\_left by 1

Method rotate\_quad(win, grid\_no, rotation):

If self.move is True and self.rotate is False:

If grid\_no is not -1:

Call self.board.rotate(grid\_no, rotation)

Set self.rotate to True

Call pygame.display.update()

Return True

1. **The algorithm for the AI moves are implemented in the “algo.py” file. We have used Genetic Algorithm for the “Easy” mode. For “Medium” mode, we have used the Minimax Algorithm with Alpha-Beta Pruning and Iterative Deepening with depth 2. We used the same algorithm for “Hard” mode but with higher depth, depth 10. We have used a Transposition Table while implementing the Minimax algorithm for optimization. We have also sorted the moves based on an evaluation function before applying minimax for further optimization. Pseudocode for this file is:**

Initialize transposition\_table as an empty dictionary

Function AlphaBeta(position, depth, max\_player, game, alpha=-infinity, beta=infinity):

pos\_key = hashable representation of position.board

If pos\_key is in transposition\_table:

Return transposition\_table[pos\_key]

If depth is 0 or position has a winner:

eval\_score = evaluate(position)

transposition\_table[pos\_key] = (eval\_score, position)

Return eval\_score, position

If max\_player:

maxEval = -infinity

best\_move = None

moves = get\_all\_move(position, WHITE, game)

moves = sorted moves by evaluate(move) in descending order

For each move in moves:

evaluation = AlphaBeta(move, depth-1, False, game, alpha, beta)[0]

If evaluation > maxEval:

maxEval = evaluation

best\_move = move

alpha = max(alpha, evaluation)

If beta <= alpha:

Break

transposition\_table[pos\_key] = (maxEval, best\_move)

Return maxEval, best\_move

Else:

minEval = infinity

best\_move = None

moves = get\_all\_move(position, BLACK, game)

moves = sorted moves by evaluate(move)

For each move in moves:

evaluation = AlphaBeta(move, depth-1, True, game, alpha, beta)[0]

If evaluation < minEval:

minEval = evaluation

best\_move = move

beta = min(beta, evaluation)

If beta <= alpha:

Break

transposition\_table[pos\_key] = (minEval, best\_move)

Return minEval, best\_move

Function iterative\_deepening(position, max\_depth, max\_player, game):

best\_move = None

For depth in range from 1 to max\_depth + 1:

\_, best\_move = AlphaBeta(position, depth, max\_player, game)

Return best\_move

Function minimax(position, depth, max\_player, game):

If depth is 0 or position has a winner:

Return evaluate(position), position

If max\_player:

maxEval = -infinity

best\_move = None

For each move in get\_all\_move(position, WHITE, game):

evaluation = minimax(move, depth-1, False, game)[0]

maxEval = max(maxEval, evaluation)

If maxEval == evaluation:

best\_move = move

Return maxEval, best\_move

Else:

minEval = infinity

best\_move = None

For each move in get\_all\_move(position, BLACK, game):

evaluation = minimax(move, depth-1, True, game)[0]

minEval = min(minEval, evaluation)

If minEval == evaluation:

best\_move = move

Return minEval, best\_move

Function Genetic\_Algorithm(board, color, game):

Constants:

POPULATION\_SIZE = 20

MUTATION\_RATE = 0.1

NUM\_GENERATIONS = 100

Function fitness(board):

Return evaluate(board)

Function create\_population(board, game):

Return get\_all\_move(board, color, game)

Function select\_population(population):

sorted\_population = sorted(population, key=lambda move: fitness(move), reverse=True)

Return sorted\_population[:POPULATION\_SIZE // 2]

Function crossover(parent1, parent2):

Return random choice between parent1 and parent2

Function mutate(move, game, population):

If random value < MUTATION\_RATE:

Return random choice from population

Return move

population = create\_population(board, game)

For each generation in NUM\_GENERATIONS:

selected\_population = select\_population(population)

new\_population = []

While new\_population size < POPULATION\_SIZE:

parent1 = random choice from selected\_population

parent2 = random choice from selected\_population

child = crossover(parent1, parent2)

child = mutate(child, game, population)

new\_population.append(child)

updated\_population = new\_population

best\_move = max(updated\_population, key=lambda move: fitness(move))

Print "Best move of genetic algorithm"

Print best\_move

Return best\_move

Function simulate\_move(move, board, game):

(row, col, grid\_no, rotation) = move

board.draw(row, col, game.turn)

board.rotate(grid\_no, rotation)

Return board

Function get\_all\_move(board, color, game):

moves = []

valid\_moves = board.get\_valid\_moves()

For each move in valid\_moves:

temp\_board = deepcopy(board)

new\_board = simulate\_move(move, temp\_board, game)

moves.append(new\_board)

Return moves

Function evaluate(board):

If check\_win(board) == 1:

Return 1000

If check\_win(board) == -1:

Return -1000

If is\_board\_full(board):

Return 0

score = 0

score += check\_4\_in\_a\_row(board, 1) \* 50

score -= check\_4\_in\_a\_row(board, -1) \* 50

score += check\_3\_in\_a\_row(board, 1) \* 10

score -= check\_3\_in\_a\_row(board, -1) \* 10

Return score

Function check\_win(board):

If AI is winner:

Return 1

Else if User is winner:

Return -1

Else:

Return 0

Function is\_board\_full(board):

Return all cells in board are not 0

Function check\_4\_in\_a\_row(board, player):

Return count of number of 4 in a row

Function check\_3\_in\_a\_row(board, player):

Return count of number of 3 in a row

## **Discussion**

In the development of the Pentago game, several core algorithms and techniques were implemented to ensure the game provides a challenging and engaging user experience. The algorithms used include Minimax with Alpha-Beta Pruning, Iterative Deepening, and Genetic Algorithms. Each of these methods plays a significant role in enhancing the AI's decision-making capability. Minimax with Alpha-Beta Pruning was used to evaluate the best possible moves by exploring the game tree. Alpha-Beta Pruning optimizes this process by eliminating branches that do not need to be explored, thus reducing the computation time. We have also used a transposition table so that the same states are not recalculated and implemented iterative deepening. The combination of these techniques allows the AI to make optimal moves within a reasonable time. Genetic Algorithm was used to make a comparatively weak AI for the easier mode of the game. This approach generates a population of possible moves, evaluates their fitness, and iteratively improves them through selection, crossover, and mutation.

## **Conclusion**

The techniques we have used resulted in a well-rounded AI opponent that can challenge human players effectively. This project demonstrates the practical application of some AI algorithms in game development, highlighting their strengths and areas for improvement. Future work could explore additional heuristics and machine learning approaches to further enhance the AI's performance and adaptability.

## **Reference**

1. https://www.ultraboardgames.com/pentago/game-rules.php
2. https://github.com/abrarhasan3/Modified-Checkers-Game