Faculty of engineering and technology.

Department of electrical and computer engineering.

Artificial Intelligence – ENCS3340.

Magnetic Cave AI Game

****

Student's name (prepared by): Amr Halahla & Jehad Halahla.

Student's ID: 1200902 & 1201467.

Data: 20 | June | 2023.

Table of contents

Table of figures

Chapter 1: Game Theory in AI

1: player vs machine in chess

Games in AI serve as valuable testbeds for developing and evaluating intelligent systems. They provide controlled environments where **algorithms** and **agents** can learn, reason, and make decisions. By designing AI agents to play games, researchers can explore and develop techniques that can be applied to real-world problems.

Games offer several advantages in AI research:

1. **Well-defined rules**: Games have clear and unambiguous rules, which make them ideal for defining goals, actions, and rewards. This simplifies the development and evaluation of AI algorithms.

2. **Competitive environment**: Games provide a competitive setting where AI agents can learn to make strategic decisions and outperform opponents. This helps in developing algorithms that can handle complex and dynamic environments.

3. **Data availability**: Games generate large amounts of data, including states, actions, and outcomes. This data can be used for training AI models and improving performance through iterative learning.

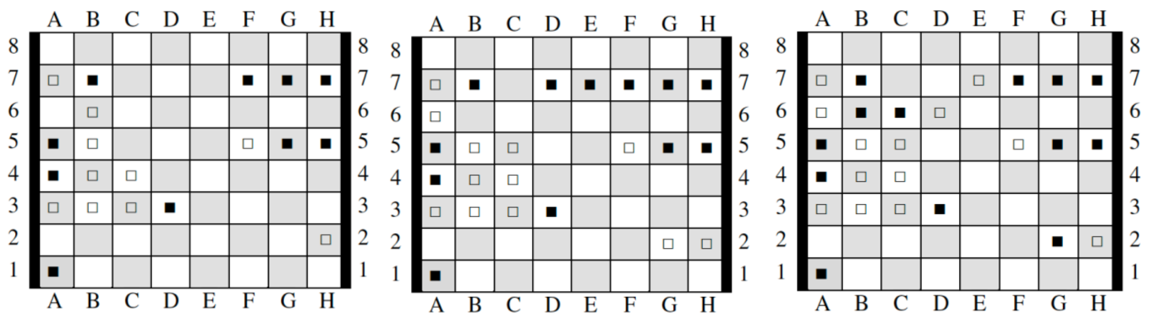
4. **Benchmarking**: Games provide a standardized benchmark for comparing the performance of different AI algorithms. By evaluating agents on common game scenarios, researchers can objectively assess progress and advancements in AI techniques.

5. **Generalization**: Successful AI agents in games often demonstrate the ability to generalize knowledge and strategies from one game to another. This showcases the potential of AI algorithms to transfer learning and adapt to new domains.

Overall, games serve as testbeds for AI research, allowing researchers to develop and evaluate intelligent systems in controlled environments. The insights gained from game-playing AI agents can be applied to real-world applications such as decision-making, resource allocation, and strategic planning.

the game we are about to design, is a good example of 2 player games, that can be analyzed and solved using AI.

Chapter 2: Magnetic Cave game



Illegal configuration:
Brick B6 cannot be placed there.

Here, player ■ wins.
He built a bridge from D7 to H7.

Here, player □ wins.
He built a bridge from A3 to E7.

2: some cases in the game

Magnetic Cave is a 2-player adversary game where each player tries to build a “bridge” of 5 magnetic bricks within a cave whose left and right walls are magnetic. For the sake of this project, the bricks of one player will be represented by a ■ and the bricks of the other by a □. The version of Magnetic Cave that you will implement will be played on a regular 8x8 chess board.

The rules of the game are simple:

- Initially, the cave (the board) is empty.

- Player ■ and player □ move in an alternate fashion, starting with ■. So ■ starts, followed by □, then ■ again, then □ again, ...

- Because there are two big magnets on each side of the cave, a player can only place a brick on an empty cell

of the cave provided that the brick is stacked directly on the left or right wall, or is stacked to the left or the right of another brick (of any color).

- As soon as one player is able to align 5 consecutive bricks in a row, in a column or in a diagonal, then this player wins the game.

- If no player Is able to achieve a winning configuration and the board is full, then the game stops and there is a tie.

Our Task

In this project, we will implement a minimax algorithm to play this game automatically.

Choosing the heuristic that we want, but our program must decide on the next move to take in at most 3 sec

Play Modes

Our program should be able to run in manual mode and in automatic mode. This means that we should be able to run our program with:

1. manual entry for both ■'s moves and □'s moves

2. manual entry for ■'s moves & automatic moves for □

3. manual entry for □'s moves & automatic moves for ■

After each move, our program must display the new configuration of the board.

Chapter 3: understanding the game

The game that was provided to us, seemed fun to play around with, and after playing the game manually for sometime we arrived at some conclusions about the game, which will help us guide our program gameplay later. Those points can be summarized as follows:

1. **Building a strong defense**: Focus on strategically placing your bricks to block your opponent's potential winning moves. Create barriers and disrupt their alignments by occupying key positions.

2. **Establishing multiple winning paths**: Aim to create multiple potential winning paths simultaneously. This increases your chances of success and forces your opponent to divide their attention (that can be done mainly by playing in the center).

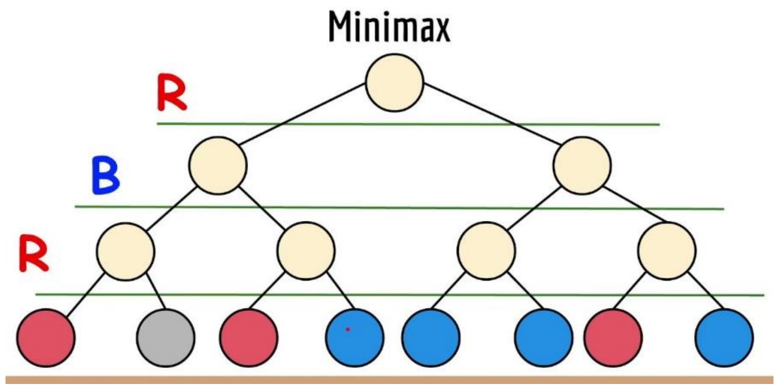
3. **Balancing offense and defense**: While defense is important, don't neglect the offensive aspect. Continuously search for opportunities to build your own winning configurations while defending against your opponent.

4. **Prioritizing high-value moves**: Identify moves that offer the highest potential for creating winning alignments. Consider both immediate gains and long-term strategies.

5. **Anticipating opponent's moves**: Try to predict your opponent's strategy and adapt accordingly. Pay attention to their placements and patterns to anticipate their next moves.

Chapter 4: Algorithms and functions

MiniMax with alpha-beta pruning:



3: Illustration of minimax

The minimax algorithm is a decision-making algorithm commonly used in game theory and artificial intelligence. Its objective is to determine the optimal move for a player in a turn-based game with perfect information.

The algorithm operates by exploring the game's decision tree, considering all possible moves and their outcomes. It assumes that both players play optimally, with one player maximizing their score (referred to as the maximizing player) and the other player minimizing their score (the minimizing player).

The minimax algorithm recursively evaluates each possible move, alternating between maximizing and minimizing players, until a terminal state is reached (e.g., a win, loss, or draw). It assigns a score to each terminal state, representing the desirability of that outcome for the maximizing player.

By backtracking through the decision tree, the algorithm propagates these scores upward, allowing each player to make the best move based on the current game state. The maximizing player chooses the move with the highest score, while the minimizing player selects the move with the lowest score.

The key idea behind the minimax algorithm is to consider the best possible moves by assuming rational play from both players. It allows us to determine an optimal strategy or select the best move in a game scenario with perfect information.

While the basic minimax algorithm guarantees an optimal outcome, it can be computationally expensive when the game has a large branching factor and depth(which applies to our case) . To mitigate this, additional techniques like alpha-beta pruning or heuristic evaluations can be incorporated to improve the algorithm's efficiency without sacrificing optimality.

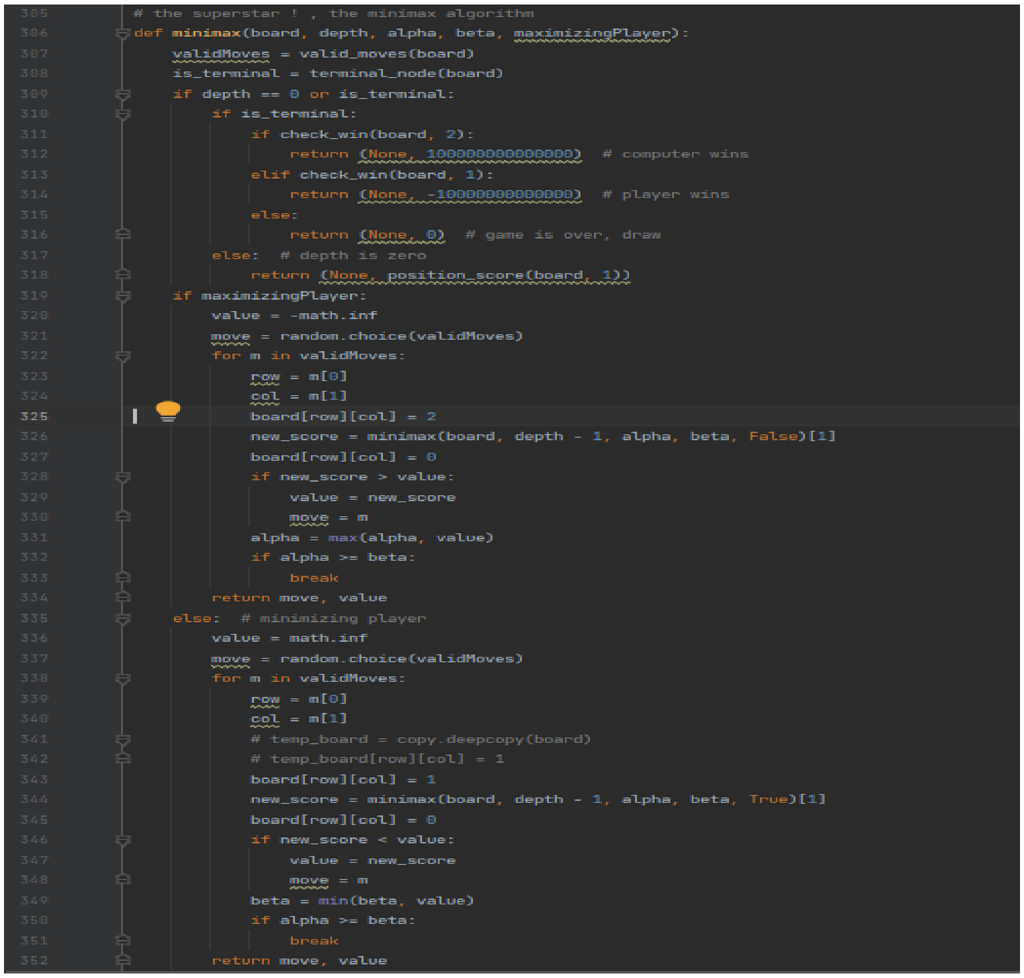
Magnetic cave analysis:

In our game the tree is expected to expand 16 times for each possible move at the start of the game, and that gives us (16^{depth}) possible states to explore and evaluate manually, for any depth for the minimax algorithm.

There are many ways to make the code run faster, other than the alpha-beta pruning technique, such as dynamic programming, or just the simplification of position score calculation heuristics.

In our implementation, we chose a complex evaluation function that uses some techniques such as: **sliding window.**

The following is our implementation of the minimax algorithm:



```python
# the superstar ! , the minimax algorithm
def minimax(board, depth, alpha, beta, maximizingPlayer):
    validMoves = valid_moves(board)
    is_terminal = terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if check_win(board, 2):
                return (None, 100000000000000)  # computer wins
            elif check_win(board, 1):
                return (None, -10000000000000)  # player wins
            else:
                return (None, 0)  # game is over, draw
        else:  # depth is zero
            return (None, position_score(board, 1))
    if maximizingPlayer:
        value = -math.inf
        move = random.choice(validMoves)
        for m in validMoves:
            row = m[0]
            col = m[1]
            board[row][col] = 2
            new_score = minimax(board, depth - 1, alpha, beta, False)[1]
            board[row][col] = 0
            if new_score > value:
                value = new_score
                move = m
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return move, value
    else:  # minimizing player
        value = math.inf
        move = random.choice(validMoves)
        for m in validMoves:
            row = m[0]
            col = m[1]
            # temp_board = copy.deepcopy(board)
            # temp_board[row][col] = 1
            board[row][col] = 1
            new_score = minimax(board, depth - 1, alpha, beta, True)[1]
            board[row][col] = 0
            if new_score < value:
                value = new_score
                move = m
            beta = min(beta, value)
            if alpha >= beta:
                break
        return move, value
```

4: minimax

In the given code the minmax evaluates the score for the tree of possible moves, and returns the best move (row, column) along with its’ score.

The position score function uses the elegant sliding window technique, and assigns score increment according to how favorable the position is, it checks negative sloped diagonals, positive sloped diagonals, rows, columns, using windows of size 5, for each of them, and also the pieces for the opponent are taken into consideration and a decrement of the score is given for the windows that contain opponent pieces and empty spots.

This score function is complicated and heavily computational, which gave us a shallow limit for the depth in order to give a good time for each move (note that the response time is reduced as we go through the game), so an ideal depth of 3 give always response time less than three seconds, and still does fairly well in gameplay.

The sliding window technique

The sliding window technique is a common algorithmic approach used to efficiently process or analyze data in a sequential manner. It involves maintaining a "window" of a fixed size that slides or moves through the data, allowing us to examine different subsets or subarrays of the data at each step.

The sliding window technique is particularly useful when dealing with problems that involve subarray or substring calculations, such as finding the maximum sum of a subarray or counting occurrences of a pattern in a string. By using a fixed-size window that moves through the data, unnecessary recalculations can be avoided, leading to improved efficiency.

The key advantage of the sliding window technique is that it reduces the time complexity of the algorithm by eliminating redundant computations. It allows us to process the data in a linear or near-linear time complexity, making it an effective approach for optimizing algorithms that deal with sequential data.

Which matches our game description perfectly, since we are trying to build sequences of 5 bricks.

Chapter 5: result analysis

In order to test our AI, we played some games against it, going second 5 times and going first 5 times too. In our game design, the AI is always blue, and the player is always red.

The program runs and we choose one of four options:

1. Player vs Player

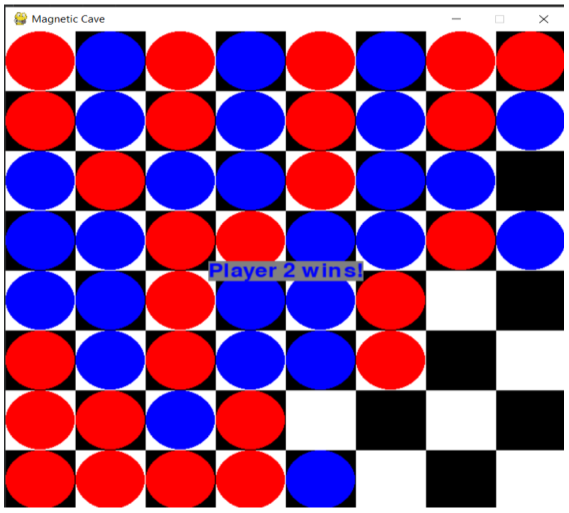
2. Player vs Computer

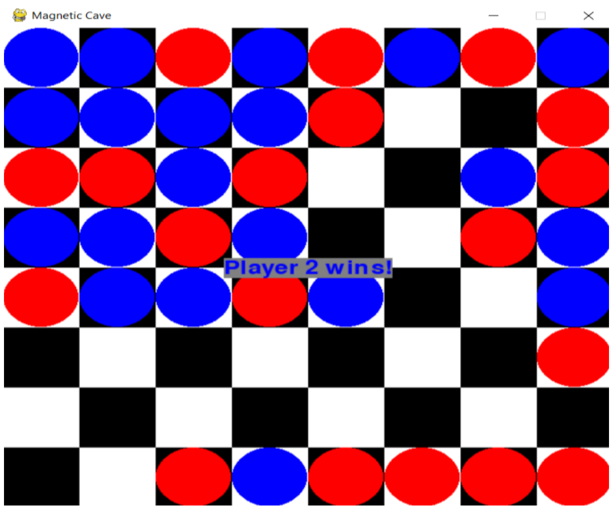
3. Computer vs Player

4. Exit

Here are some screens for the gameplay screen:

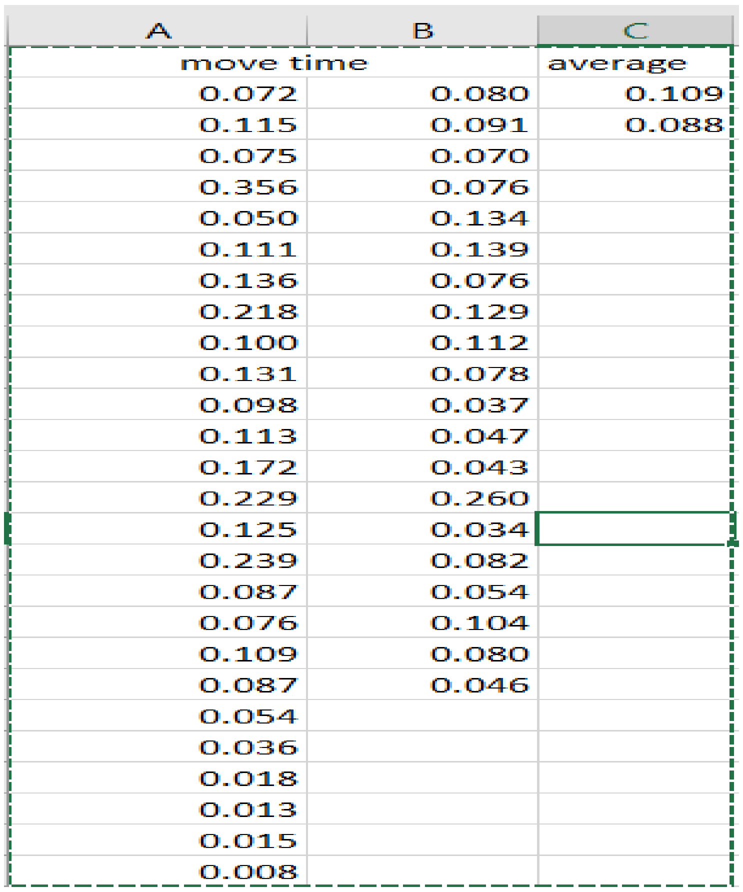
sample games with Depth = 3





5: game play screen, depth 3, mode 2 & 3

Time Taken for two sample games with their average (in seconds):



A	B	C
move time		average
0.072	0.080	0.109
0.115	0.091	0.088
0.075	0.070	
0.356	0.076	
0.050	0.134	
0.111	0.139	
0.136	0.076	
0.218	0.129	
0.100	0.112	
0.131	0.078	
0.098	0.037	
0.113	0.047	
0.172	0.043	
0.229	0.260	
0.125	0.034	
0.239	0.082	
0.087	0.054	
0.076	0.104	
0.109	0.080	
0.087	0.046	
0.054		
0.036		
0.018		
0.013		
0.015		
0.008		

6:table of move times

Thanks

Thanks for our instructors of the course, Dr.Yazan Abu Farha and Dr. Ismail Khater.

That was an exciting and educational project to work at, we hope the second project is exiting as this one.

For who ever is reading this report, please check out github pages:

https://github.com/jehad-halahla

https://github.com/Amr-HAlahla