Zero Point One

Maria Rabelo up202000130
Máximo Pereira up202108887
Edoardo Tonet up202311494

What is Zero Point One?

0.1 is a 2 player board game that merges pieces and ideas from many other ones, such as Chess or Shogi

It is played on a 8x8 board and each players begins with 16 pieces, labeled with pairs of numbers (0.1, 0.2, 1.1, 1.2, 2.2)

These labels also describe those pieces’ movement ranges

It is possible to bring captured pieces back to life.

The game ends when the opponent 0.1 piece is captured.

Formulation as a Search Problem

State Representation

board, currentPlayer, stateValue, capturedPieces

Initial State

An 8x8 matrix with red pieces occupying the two rows at the bottom and blue pieces occupying the last two rows at the top. The first player is red.

Objective Test

A function that returns the winner if the 0.1 piece has been captured or a draw if the game is still in progress.

Formulation as a Search Problem

Operators

Both options cost one turn.

move_piece

piece in original pos must be currentPlayer`s,

destination pos must not have piece belonging to currentPlayer

piece will be moved to destination

recover_piece

piece to be recovered must initially not be currentPlayers`s

position to set piece must be empty

Evaluation function

Assigns a score based on the game state. This value will be used by the Minimax algorithm for choosing the best movement option.

Implementation Approach

Evaluation Function

Quantitatively assesses game states to guide AI towards winning scenarios.

```
def evaluate(self):
    score = 0
    for row in self.board:
        for cell in row:
            if cell.startswith(self.current_player):
                score += 1  # Favourable to the current player
            elif cell != ' -- ':
                score -= 1  # Favorable to the opponent
    return score
```

Operators

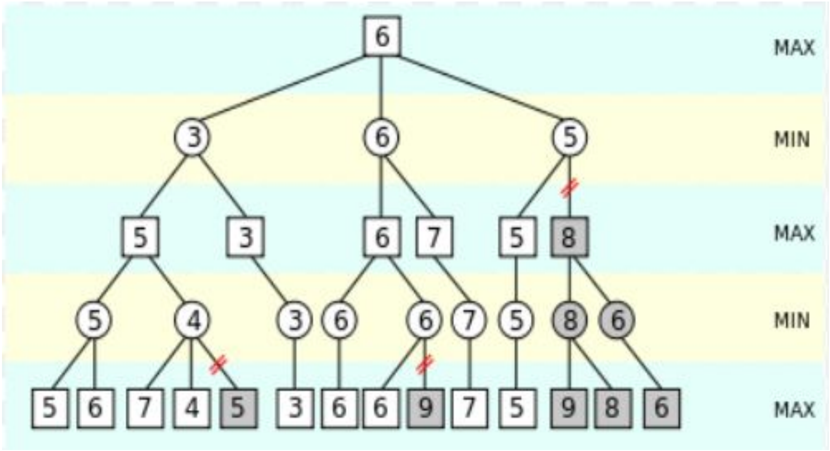
```
def make_move(self, move):
    (start_row, start_col), (end_row, end_col) = move
    if self.board[end_row][end_col] != ' -- ':
        self.capturedPieces.append(self.board[end_row][end_col])
    self.board[end_row][end_col] = self.board[start_row][start_col]
    self.board[start_row][start_col] = ' -- '
    self.switch_current_player()

def recover_piece(self, recovery):
    choosen_piece, (dest_row, dest_col) = recovery
    for piece in self.capturedPieces:
        if piece == choosen_piece:
            self.capturedPieces.remove(piece)
            break
    new_piece = self.current_player + choosen_piece[1:]
    self.board[dest_row][dest_col] = new_piece
    self.switch_current_player()
```

The Algorithm behind our AI

For the development of our ZeroPointOne AI, we opted for a MiniMax algorithm, since, after various tests, it turned out to be the most efficient one for an adversarial search situation;

Even though after the first implementation the algorithm was working correctly, we noticed that the time spent was excessive for certain depth values, so with the help of Alpha-Beta cuts and iterative deepening to improve the performance, we managed to greatly improve the AI's performance.



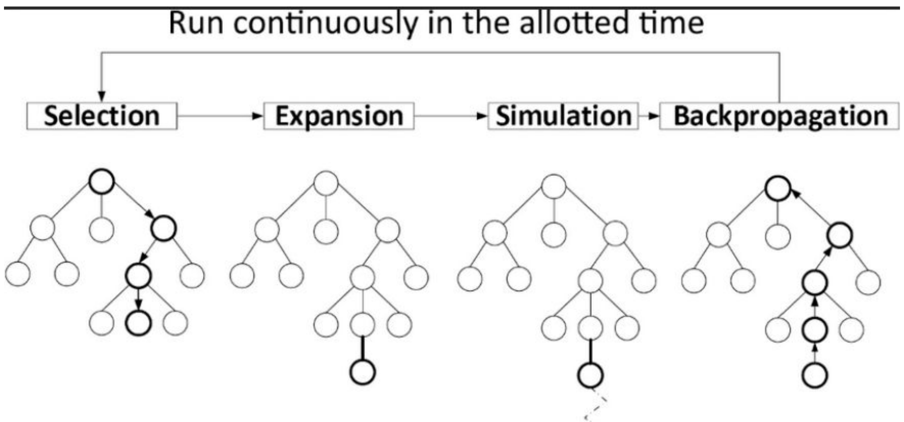
Example of Minimax algorithm with Alpha-Beta cuts

Algorithm choice

During the implementation of the AI we also wondered the possibility of using a Monte Carlo Tree Search algorithm, being a universally applicable algorithm with no need of an evaluating function; although after some tests we noticed that, even if faster, the moves were rarely smart;

This brought us to the conclusion that the MiniMax algorithm would be more accurate in such situation even if with a limited depth; our choice was made to give priority to accuracy and solidity, while trying to maintain an efficient computational time expense, which was reached with Alpha-Beta Cuts, iterative deepening and a valid evaluating function.

Minimax algorithm was also chosen in order to give players the possibility to choose the difficulty of the game, based on the AI's searching depth.



Monte Carlo Tree Search working method

Conclusion

AI Algorithm:

After testing, the Minimax algorithm with Alpha-Beta pruning and iterative deepening was chosen for its efficiency and accuracy in adversarial search situations.

Despite considering the Monte Carlo Tree Search (MCTS) for its applicability without an evaluation function, it was not chosen due to lower move quality.

Performance Improvements:

Alpha-Beta pruning and iterative deepening significantly enhanced the AI's performance, balancing computational time with move accuracy.

The AI's ability to choose difficulty levels through depth adjustment offers players a customizable challenge.

Insights and Learnings:

The choice of algorithm plays a crucial role in the balance between move quality and computational efficiency.

Careful formulation of the game as a search problem is crucial for the AI's success.

The iterative approach to development and testing allowed for the refinement of AI performance and player experience.

Future Directions:

Exploring more advanced heuristics for the evaluation function could further enhance the AI's decision-making.

Investigating other AI algorithms or combinations thereof might offer new strategies and efficiencies.

Implementation work and References

Programming Language

Python

Development Environment

VSCode

Data Structures

List of lists for representing the rows and columns of the board

List for representing the captured pieces

Web pages references:

https://boardgamegeek.com/boardgame/114307/01-zero-point-one

https://boardgamegeek.com/thread/799045/complete-rules-link-pdf-and-diagrams

Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks

ML | Monte Carlo Tree Search (MCTS) - GeeksforGeeks