ABOYNE

Adversarial Search Methods

UP202108872-GUILHERME COUTINHO

UP202108895-XAVIER OUTEIRO

UP201706105-MIGUEL FIGUEIREDO

GAME DEFINITION

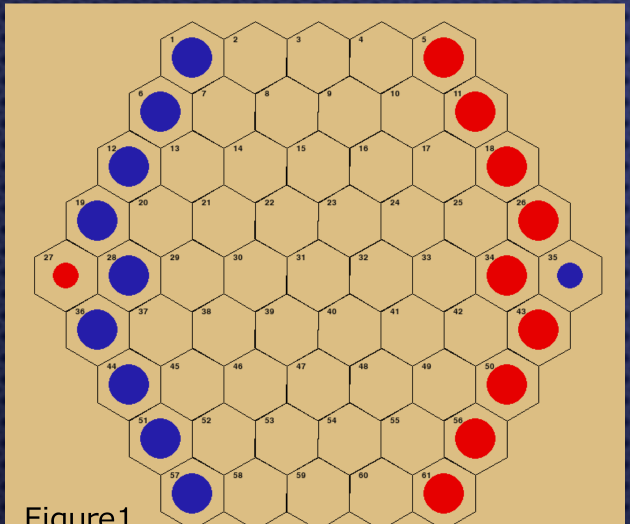


Figure1

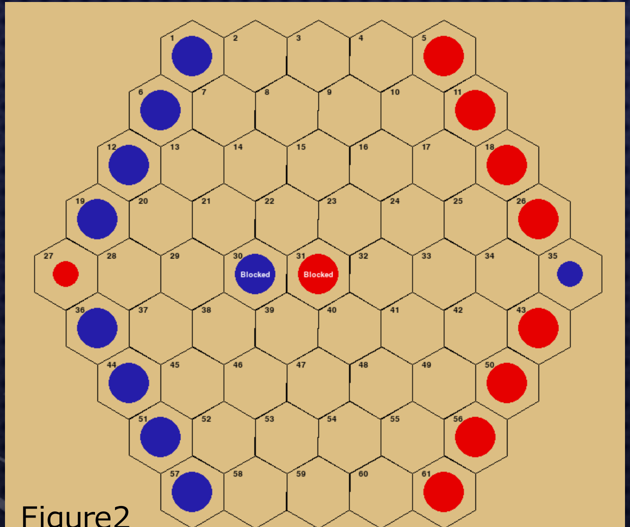


Figure2

- Aboyne is a two player board game where each player takes turns moving their pieces trying to get into their respective goal cell, in Figure1 we can see the initial board
- Moving a piece into a position next to an enemy piece will block both player's pieces
- At each turn, each player must move one of his non-blocked pieces:
- A piece may move to an adjacent empty cell or jump over a line of friendly pieces landing on the immediate next cell. If that cell is occupied by an enemy piece, that piece is captured.
- A piece cannot move into the opponent's color goal cell.
- The game can end if a player gets to their goal cell or if you block all of the enemy player's pieces

STATE REPRESENTATION

Game State

The game state is represented by all pieces on the board, where each piece has as attributes their color, their position and if its blocked or not, for each piece we can infer a list of valid moves before executing the function movePiece, which will take a piece and check if a valid move is found on nearby_hexagons.
The game state also has the current player's turn.

Initial State

The initial state is always the same the pieces start at their initial position , only the player's turn can differ at the initial state, because it is randomized at each game.

Objective Test

The game ends when a player has all its pieces blocked, resulting on that player losing. Or if a player is able to move a piece to their own goal cell, in that case this player wins.

OPERATORS

- Move operator - The move operator can be represented as a tuple(p,n) where p refers to the piece being moved and n as the new position on the board where piece will be placed.
  For the operator to work p can't be blocked and the n can't be an enemy piece, only if it involves the capture operator.

- Capture operator - The capture operator is an extension of the move operator, represented as a tuple(p,n) where p refers to the piece being moved and n as the new position where there is a enemy piece that's going to be captured
  For the operator to work p can't be blocked and n is occupied by an enemy piece where in the vector of p->n there should be a player piece in between both positions.

- Block operator - The block operator can be represented as a tuple(p,h) where p is a piece that's potentially blocked and h are the nearby hexagons around the piece, if on a nearby hexagon exists an enemy piece then both pieces get blocked.
  For the operator to work h must have an enemy piece.

MINIMAX WITH HEURISTICS/EVALUATION FUNCTION

To create different bot levels, we will vary the depth of a Minimax search. Higher depths will allow the bot to explore more possible future states, resulting in a more challenging gameplay.

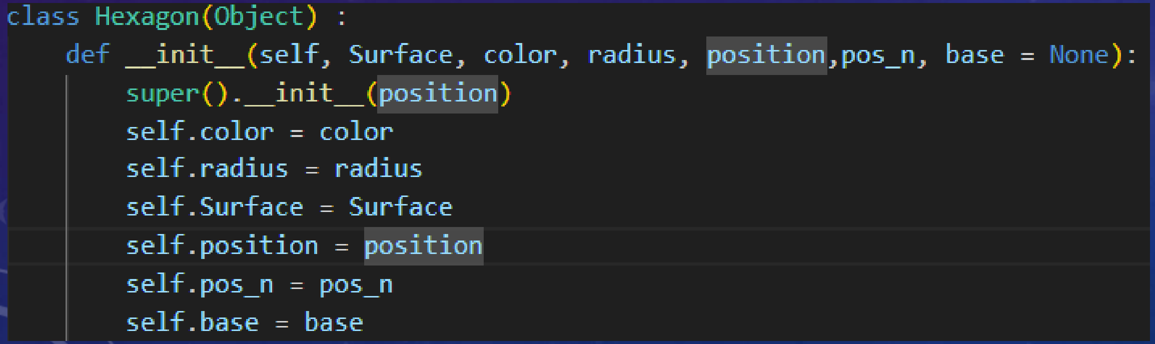
In addition to varying the depth of the Minimax search, we can enhance the bot's decision-making by incorporating an evaluation function. This function assigns a numerical value to each game state, representing the desirability of that state for the current player

We haven't discussed how the evaluation function will work yet, but these are some factors we will take in account when creating it for a given state :

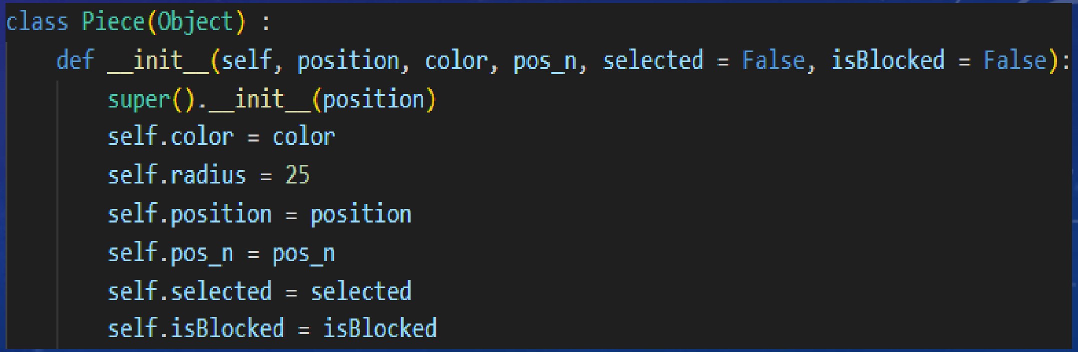
- Number of non blocked pieces each player has
- State of the piece, if its blocked or not
- Distance of each player's stones to their respective goal cells
- Potential to capture an enemy's piece
- Number of possible moves each player has (better mobility = better position)

CHECKPOINT

- At this point we have the game fully functional in a player vs player mode.
- The game is being developed in python, using pygame library, and our approach was to make everything an object, so we have 2 main classes, Hexagon and Piece, which extend Object.
- Before the game starts the board is drawn, populating an array, *hexagons,* with the necessary instances of the Hexagon class, and, also two other arrays, *blue_pieces* and *red_pieces,* with 9 instances, for each color, of the Piece class.



```
class Hexagon(Object) :
    def __init__(self, Surface, color, radius, position,pos_n, base = None):
        super().__init__(position)
        self.color = color
        self.radius = radius
        self.Surface = Surface
        self.position = position
        self.pos_n = pos_n
        self.base = base
```



```
class Piece(Object) :
    def __init__(self, position, color, pos_n, selected = False, isBlocked = False):
        super().__init__(position)
        self.color = color
        self.radius = 25
        self.position = position
        self.pos_n = pos_n
        self.selected = selected
        self.isBlocked = isBlocked
```

EVALUATION FUNCTION

- Block/Unblock - A piece that is blocked has a score of 20 while an unblocked piece has a score of 100.
- Goal cell - A piece reaching the goal cell receives an infinite score, ensuring that such moves are always chosen.
- Distance to goal cell - The score of a piece is determined by its distance from the goal cell, with closer proximity resulting in a higher score. Additionally, the score accounts for whether the piece is blocked or not. If the piece is blocked, the value of the distance is reduced to one-third of its original value.

The score that is used to evaluate the value of a possible move is:

Score = evaluateFunction(allPlayerPieces) - evaluateFunction(allEnemyPieces)

IMPLEMENTED ALGORITHMS:

We implemented the Minimax Algorithm using several key functions that work together to construct the game tree, evaluate possible moves, and select the best move for the AI player, these are the functions :

buildTreeMiniMax && evolveMove - recursively constructs the game tree by generating all possible moves for the current state of the game, according to the depth that is passed as an argument. In the leafs of the tree, the current state of the board is evaluated with the **evaluateGame** and those values are moved up the tree, according to the minimax algorithm. It also cuts the nodes that, according to the alfa&beta prunning algorithm, won't be chosed, improving the performance of the algorithm.

evaluateGame - updates the score for each piece and the scores of both players

minimax – having the full tree built, this functions only choses from the depth 0 nodes, the one that contain the best score of all possible moves.

This algorithm is used in the different game modes for the AI player, in AI vs AI and Player vs AI modes, and is used to give hints to the Players, in Player vs Player and AI vs Player mode. In the case of the hints the depth of the tree is 1. Otherwise is up to the user to choose if they want the AI to be in Easy (Depth 1), Medium (Depth 2) or Hard (Depth 3) difficulty.

PERFORMANCE

- In terms of performance, even though we implemented Alpha-Beta pruning in the minimax algorithm, we weren’t able to optimize it sufficiently for the hard difficulty mode (Depth 3). On average, it takes 45 seconds to make a move in this mode. The Medium difficulty takes around 25 seconds, while the Easy mode completes moves in no more than 2 seconds.

- We believe that the average branching factor of 4 for each piece, combined with 9 pieces for each player, contributes to this issue. Additionally, we regret using Python for this part of the project. If we had chosen a language like C/C++, we could have significantly improved performance.

CONCLUSION

- This project was very challenging due to the fact of having to implement something different than what we are used to, but in the end it was very rewarding and we have enjoyed seeing that the AI player was actually playing better than us.
- It was also interesting to put in practice what we learn in theory and have another perspective on how the minimax algorithm works; we think that it helped us consolidate the knowledge that we had obtained before.
- Finally, we would've wanted to implement the Monte Carlo search tree solution, but due to the workload of other curriculum units, we weren't able to go through with it.

REFERENCES

- Aboyne
- Classes Slides – Adversarial Search