Projeto de Inteligência Artificial

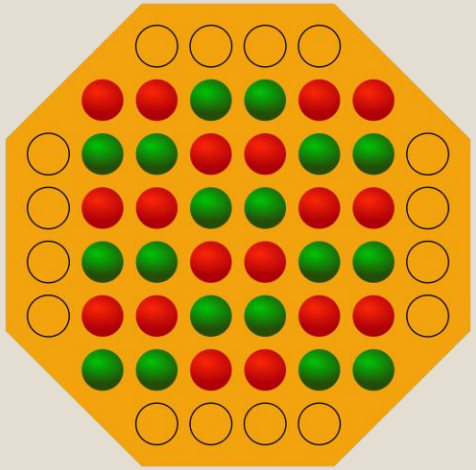
FOCUS

Turma 13 - Grupo A1_76

João Miguel da Silva Lourenço (up202108863@up.pt)
Tiago Ribeiro de Sá Cruz (up202108810@up.pt)
Tomás Filipe Fernandes Xavier (up202108759@up.pt)

1. Definition of the Game

- Focus is an abstract strategy board game. In this version, two players move stacks of one to five pieces around a checkerboard in a hexagonal format.
- The game starts with the board already filled with pieces in the following format:



- Stacks may move as many spaces as there are pieces in the stack. Players may only move a stack if the topmost piece in the stack is one of their pieces. When a stack lands on another stack, the two stacks merge; if the new stack contains more than five pieces, then pieces are removed from the bottom to bring it down to five. If a player's own piece is removed, they are kept and may be placed on the board later in lieu of moving a stack. If an opponent's piece is removed, it is captured. The last player who is able to move a stack wins.

2. Related Work

- We can see a quick explanation of the game in Wikipedia https://en.wikipedia.org/wiki/Focus_(board_game) and we can see Hasbro's official PDF with the rules for their version of this game https://www.hasbro.com/common/instruct/domination.pdf.
- We could also find various GitHub repositories with an implementation of the game, such as this pygame implementation https://github.com/fgaTactics/tecprog2017.1/blob/devel/gameEngine/GameEngine.py.
- We could not find an implementation of the game that also implements an AI that uses the minimax algorithm.

3. Formulation of the Problem as a Search Problem

- State Representation
  - Board
- Initial State (board of 6 by 6 with stack size of 3)
  - 0b111111000000000000000000000000111111000000000001000001000010000010000000000000000001000001000000100000100000000000000000100000100001000001000000000000000001000001000000100000100000011111100000000000000000000000111111
- Objective tests
  - Player can move the stack and the opponent can’t.

Operators

Name	Preconditions	Effects	Costs
move_up (n,cell,board,player)	cell.topmost_piece_color = player.color, cell.stack_size >= n, board.is_inside_board (cell.row – n)	new_stack = old_stack + moved_stack, if new_stack_size > board.stack_max_size then remove_bottom_pieces(new_stack_size - board.stack_max_size), if removed_pieces = player.color then keep_for_later else capture	1
move_down (n,cell,board,player)	cell.topmost_piece_color = player.color, cell.stack_size >= n, board.is_inside_board (cell.row + n)	new_stack = old_stack + moved_stack, if new_stack_size > board.stack_max_size then remove_bottom_pieces(new_stack_size - board.stack_max_size), if removed_pieces = player.color then keep_for_later else capture	1
move_left (n,cell,board,player)	cell.topmost_piece_color = player.color, cell.stack_size >= n, board.is_inside_board (cell.column – n)	new_stack = old_stack + moved_stack, if new_stack_size > board.stack_max_size then remove_bottom_pieces(new_stack_size - board.stack_max_size), if removed_pieces = player.color then keep_for_later else capture	1
move_right (n,cell,board,player)	cell.topmost_piece_color = player.color, cell.stack_size >= n, board.is_inside_board (cell.column + n)	new_stack = old_stack + moved_stack, if new_stack_size > board.stack_max_size then remove_bottom_pieces(new_stack_size - board.stack_max_size), if removed_pieces = player.color then keep_for_later else capture	1
place_piece_from_stack(cell,board,player)	player.personal_stack.size > 0 board.is_inside_board (cell)	new_stack = old_stack + player.color, if new_stack_size > board.stack_max_size then remove_bottom_pieces(new_stack_size - board.stack_max_size), if removed_pieces = player.color then keep_for_later else capture	1

4. Work carried out so far

- To develop this game, we decided to use Python and the library Pygame. Python provides a lot of libraries and built-in functions that aid development, Pygame being one of them.
- Pygame is an especially useful library since it provides us with several functions that are very useful for this project, since it's a library aimed at game development.

Architecture

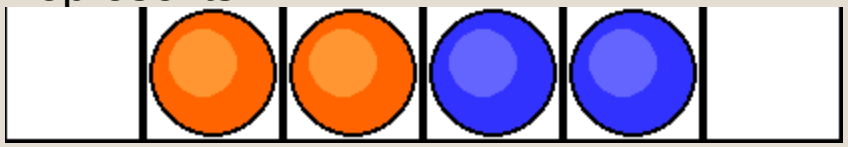
We decided to use the MVC architecture to develop this game, which stands for Model-View-Controller:

- The **model** classes take care of the game logic, such as placing a piece or moving a stack.
- The **view** classes take care of the visual aspect of the game, such as drawing the board or a piece.
- The **controller** classes handle different input events, such as clicking a piece.

One change/optimization from the previous checkpoint was representing the board as a bitboard.
In this representation, a pair of bits represents the value of a piece in a stack:
In the initial state of a board of 6 by 6 with the size of a stack of 3:
000000|000001|000001|000010|000010|000000

- `PIECE_NONE = 0b11 (this represents the corners cutoff from the board)`
- `PIECE_EMPTY = 0b00`
- `PIECE_ORANGE = 0b01`
- `PIECE_BLUE = 0b10`

Represents:



The Medium Bot evaluation function:

```
330     def eval_medium(self, state, depth) -> int:
331         if state.verify_win():
332             return 1000 + depth
333         current_player = state.get_current_player()
334         next_player = state.get_next_player()
335         return len(current_player.get_cells()) - len(next_player.get_cells())
```

This evaluation function only compares the **cells** each player has. These **cells** represent the stacks the player can move, meaning how many stacks have the topmost piece belonging to the player. This is a relatively good indicator of the state of the game, but fails to consider many other variables, therefore starts getting worse when increasing the depth.

The Hard Bot evaluation function:

```
344     def eval_hard(self, state, depth) -> int:
345         if state.verify_win():
346             return 10000 + depth
347 
348         current_player = state.get_current_player()
349         next_player = state.get_next_player()
350 
351 
352         cells_difference = len(current_player.get_cells()) - len(next_player.get_cells())
353         controlled_cells_difference = len(current_player.get_controlled_cells()) - len(next_player.get_controlled_cells())
354         stack_difference = current_player.get_stack_count() - next_player.get_stack_count()
355 
356         hidden_enemy_pieces = next_player.get_total_pieces() - len(next_player.get_cells()) - next_player.get_stack_count()
357 
358         total_pieces_difference = current_player.get_total_pieces() - next_player.get_total_pieces()
359 
360         return cells_difference + 2*controlled_cells_difference + stack_difference + 3*total_pieces_difference + hidden_enemy_pieces
```

This expands upon the other evaluation function by also comparing the controlled cells, meaning cells the player can move to, personal stack's size, total number of pieces on the board and also how many opponent's pieces are hidden, meaning they are under a stack and can't move. We also attribute different weights to each factor, based on what improved the bot's chances of winning.

We have implemented 4 different bots:

- Easy bot
- Medium bot
- Hard bot
- MCTS bot

While the easy bot plays random moves and the MCTS bot uses the Monte-Carlo Tree Search algorithm, the Medium bot and Hard bot both use the **Negamax** algorithm. We decided to use the **Negamax** algorithm because its properties apply very well to our project due to the nature and mechanics of the **Focus** game, seeing as it is a zero-sum game.

Expanding upon the **Negamax** algorithm, to improve its performance, not only is the bit representation of the board crucial, but also the memoization we implemented as well. Inside every call to **Negamax,** after processing every valid move and choosing the best one, we call the **add_to_memo()** function. This function stores in a dictionary the relevant information about the state after a certain move and its respective value. Additionally, we also add to the dictionary the value of the mirrored version of the game state and the state from the view of the other player. The mirrored version simply represents a position where the board seems to be "mirrored". Although the pieces have changed their absolute position, the relative position between every piece stays the same. Because of this, the value of a mirrored board is the same as a normal board. The other game state we add is one where the board stays the same, but the players switch positions. This state has the negative value of the original state. Finally, we also add the mirrored version of this other state, also with negative value, because mirroring does not affect the state of the game.

Therefore, after visiting a state, we can compute and store the value of 4 states in the dictionary. This boosts performance significantly, especially since this dictionary is preserved between games.

Results

Medium Bot vs Hard Bot

Bot	Avg, Time per Move	Avg, States Considered	Avg, States Avoided	Avg, Branches Pruned	W/L Ratio	Avg, Total Moves
Medium Bot	0,113649	6232	1415	2034,8	0%	9,2
Hard Bot	0,51491			2044,6	100%	

Easy Bot vs Hard Bot

Bot	Avg, Time per Move	Avg, States Considered	Avg, States Avoided	Avg, Branches Pruned	W/L Ratio	Avg, Total Moves
Easy Bot	2,39E-05	4210	6,7	0	0%	8
Hard Bot	0,951371			3294,6	100%	

MCTS vs Hard Bot

Bot	Avg, Time per Move	Avg, States Considered	Avg, States Avoided	Avg, Branches Pruned	W/L Ratio	Avg, Total Moves
MCTS Bot	0,721265	3645	6,5	0	0%	8,7
Hard Bot	0,770966			2866,6	100%	

Easy Bot vs Medium Bot

Bot	Avg, Time per Move	Avg, States Considered	Avg, States Avoided	Avg, Branches Pruned	W/L Ratio	Avg, Total Moves
Easy Bot	3,535E-05	40454	13958	0	0%	55
Medium Bot	0,29866644			24244,9	100%	

Conclusions

- With this project, something important that we concluded was that, in the context of adversarial games, regarding the Minimax algorithm and its variants, a good evaluation function is crucial for the bot to pick the correct move. This may seem obvious, but something we struggled with was having a very good evaluation function at low depths, but, when increasing the depth, the "intelligence" of the bot decreased significantly. This is due to the fact that a subpar evaluation function is very noticeable at high depths, because the relatively bad move choices trickle down into the choice tree and make the bot repeatedly choose moves that don't necessarily benefit him, in the context of the game.