Giorgio Mongardi **Report**

Computational Intelligence 2023/24

A delayed start

At the start of october i started to follow the first lessons in presence.
I would have continued, but unfortunately i had to stop for a while.
I started to catch up with the lessons towards the end of November.

Catching up

At the time i was already aware that I would have lost part of the experience of the course because i was behind, so I decided to exercise both on labs and on examples from the slides while watching the recorded lessons.

In the last week of November i tried and solved:

- *the friend-pizza problem*: trying not to copy the solution the professor gave, in particular mine was different because it considered the bike, adding to the representation where the bike was and which people had onboard;
- *the set covering problem*: with single state method trying different fitnesse and tweak functions, with path search algorythms;
- *Lab 1*: trying different possibilities for the *alpha star evaluation function*;
- *the klotski puzzle*: using a similar approach to the one used for the pizza-friend problem;
- *the 15kg problem*: using path search strategies.

After watching some more lessons i exercised with the *Halloween challenge*, obtaining my best solution be evaluating the contribution each possible new entry would introduce and chosing the best one.

Then i started working on nim (Lab2), creating an agent able to play the game. First I implemented the expert agent based on the knowledge the professor gave us: that the nim-sum (bitwise xor of the rows) for a winning move would result in 0 and that who starts with a zero nim-sum should always win if it takes only winning moves. Then I implemented an ES assigning to each state a move. It learned to win starting from a winning position against the expert player, but couldn't win when starting from losing positions.

While I was working on these implementations, I noticed that there are some cases for which the considerations made for the expert model fail: when the number of rows with more than 1 element is one or zero, then the optimal strategy isn't to search for the zeroing of the nim-sum, but became to try to obtain an even number of rows to 1 element.
One example is the case in which we have the board in state 1-1-2: if the player tries to obtain a zero nim-sum it will obtain state 1-1, which is winning for the opponent, while if it takes one pin from 2 then it will obtain state 1-1-1, which, even if it doesn't have a zero nim-sum, it's a winning state.

After that i also did Lab9.

Finally caught up

Before the christmas holidays i finally caught up with the lessons, so this time i knew that i could submit the assigned Lab10 in time.

Between a Christmas party and the other, i worked on Lab10 and wrote my first implementations of *Minimax* and *Monte Carlo* in order to create an agent able to play tic-tac-toe against a random opponent.
For MC, i tried training both against a random opponent and against itself.
I also tried to add symmetries, but as i now now, making some mistakes.

After that i kept following the lessons, while working on other exams.

Project implementation start

I started working on the project on the 20th of january.

The first thing that I did was to analyze the problem at hand, to do that I played a little around the Game class provided by the professor.

In this part i didn't know if I could modify the Game class as I wanted or not, so I decided to leave it as it was to avoid problems. I wasn't trying any specific strategy, so I wrote a simple deep model trained in an evolutionary way just to better comprehend quixo as a game, the Game class and which moves were considered legal.

This part was useful: I learned that some moves that are legal in the real quixo aren't in this implementation (removing a tile and pushing it back in, Ex: *((0, 0), Move.TOP)*) and that I would have to make some modifications in order to have informations on the single moves performed.

While playing around defined some bases to work on:

- I started pre-computing all legal moves to save time, without considering occupied tiles:

```
border = []
for i in range(5):
    for j in range(5):
        if i == 0 or i == 4 or j == 0 or j == 4:
            border.append((i, j))
BORDER = (list(set(border)))
print(len(BORDER))

def tile_to_moves(tile):
    possible_moves = [Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT]

    if tile[0] == 0: possible_moves.remove(Move.LEFT)
    if tile[0] == 4: possible_moves.remove(Move.RIGHT)
    if tile[1] == 0: possible_moves.remove(Move.TOP)
    if tile[1] == 4: possible_moves.remove(Move.BOTTOM)

    return possible_moves

tile_moves = {tile: tile_to_moves(tile) for tile in BORDER}
```

```
ALL_MOVES = []
for tile in BORDER:
    possible_moves = tile_moves[tile]
    for move in possible_moves: ALL_MOVES.append((tile, move))
N_ALL = len(ALL_MOVES)
```

- I tried to consider symmetries (making an error on the implentation that I didn't recognized right away);
- I wrote a first skeleton for the evolutionary strategy.
- I considered the possibility of using a number as state instead of a tuple:

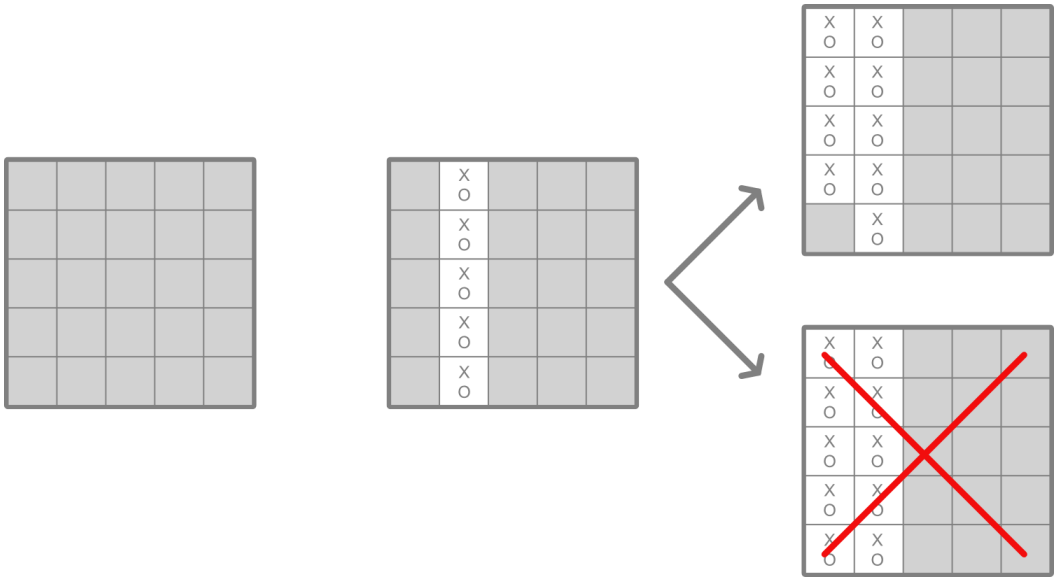
```
def state_to_board(state):
    binary_string = format(state, '050b')
    binary_array = np.array(list(map(int, binary_string))).reshape(2, 5, 5)
    board = np.zeros((5, 5), dtype=int)
    board[binary_array[0] == 1] = -1
    board[binary_array[1] == 1] = 1
    return board

def board_to_state(board):
    binary_array = np.zeros((2, 5, 5), dtype=int)
    binary_array[0][board == -1] = 1
    binary_array[1][board == 1] = 1
    binary_string = ''.join(map(str, binary_array.flatten()))
    return int(binary_string, 2)
```

Starting with actual methods

After a couple of days I had an acceptable comprehension of the problem and I started using what I found to start the implementation of *Minimax* and *Monte Carlo*.

Here I already knew that there were a lot of possible states in Quixo, but I overlooked it. It was when I was trying to compute all states that the actual size of the problem became clear to me.



If we count as legal those moves that allow for a player to leave the board unchanged, then the number of possible states should be 3^{25}.
Winning states block some state from happening: when there is a winning line on the board, there cannot be another. For each vertical or horizontal line there are 3^{15} impossible states. If my reasoning is correct then there should be something like 3^{25} - (10 * (4 *3^{15})) possible states in Quixo, with 12 * 3^{20} – (10 * (4 *3^{15})) possible winning states.

Having understanded this issue, I began to implement the two algorythms considering new states only when needed using a dictionary.

I won't lie, the implementations took me some time: I found myself to be relatively slow at writing code and searching for bugs in my implementation. This is probably because i started writing code without having first designed the algorythms on paper.

During this time i found out the mistake i made on symmetries (in particular on how to reverse the moves), so I resolved it:

```
dict_rot = {(Move.TOP, 1): Move.RIGHT, (Move.TOP, 2): Move.BOTTOM,
            (Move.TOP, 3): Move.LEFT,(Move.BOTTOM, 1): Move.LEFT,
            (Move.BOTTOM, 2): Move.TOP, (Move.BOTTOM, 3): Move.RIGHT,
            (Move.LEFT, 1): Move.TOP, (Move.LEFT, 2): Move.RIGHT,
            (Move.LEFT, 3): Move.BOTTOM, (Move.RIGHT, 1): Move.BOTTOM,
            (Move.RIGHT, 2): Move.LEFT, (Move.RIGHT, 3): Move.TOP
```

```
dict_flip = {Move.TOP: Move.TOP, Move.BOTTOM: Move.BOTTOM,
             Move.LEFT: Move.RIGHT, Move.RIGHT: Move.LEFT}

#rot_orario: (3, 4) -> (4, 1) -> (1, 0) -> (0, 3) -> (3, 4)
#: (xi, yi) -> (yi, 4 - xi)
#rot_anti_orario: (3, 4) -> (0, 3) -> (1, 0) -> (4, 1) -> (3, 4)
#: (xi, yi) -> (4 - yi, xi)

def rot(n_rot):
    def rot_n(from_pos, move):
        for _ in range(n_rot):
            from_pos = 4 - from_pos[1], from_pos[0]
        return from_pos, dict_rot[(move, n_rot)]
    return rot_n

def flip(from_pos, move):
    from_pos = 4 - from_pos[0], from_pos[1]
    return from_pos, dict_flip[move]

def flip_rot(n_rot):
    def flip_rot_n(from_pos, move):
        from_pos, move = rot(n_rot)(from_pos, move)
        return flip(from_pos, move)
    return flip_rot_n

inverse_simmetries =
[rot(1),rot(2),rot(3),flip,flip_rot(1),flip_rot(2),flip_rot(3)]

# I omitted check_simmetries(board, state_list) -> base_state, id_simmetry

MOVES_SIMMETRIES = {} #(id_move, id_simmetry) -> id_move
for id_move in range(len(ALL_MOVES)):
    from_pos, move = ALL_MOVES[id_move]
    for id_simmetry in range(len(inverse_simmetries)):
        idx = None
        for i in range(len(ALL_MOVES)):
          if ALL_MOVES[i] == inverse_simmetries[id_simmetry](from_pos, move):
                idx = i
                break

        MOVES_SIMMETRIES[(id_move, id_simmetry)] = i
```

For the first implementations of *Minimax* and *Monte Carlo* I used a modified version of the Game class that allowed to perform single moves.

New implementations

At the end of the second day I decided to restart with the implementations from scratch, this time I designed them on paper before writing the code.

I also decided to use a Dummy_Game class to train my agents while leaving the original Game class untouched and use it on tests.

```
class Dummy_Game(object):
    def __init__(self) -> None:
        self._board = np.ones((5, 5), dtype=np.uint8) * -1
        self.current_player_idx = 1
    def single_move(self, board, from_pos, move, player_id):
        self._board = deepcopy(board)
        self.current_player_idx = player_id
        ok = self.__move(from_pos, move, player_id)
        return deepcopy(self._board), ok
    def check_winner_board(self, board):
        self._board = board
        return self.check_winner()

    ## then the other functions of the original Game class
```

The next day i wrote the code for both and, as expected,
I encountered much less problems then the first time.

I keep talking about *Monte Carlo* and *Minimax* toghether because i wrote them alternating from one to the other so that while the first was running I could wrote the code for the second (probably not the best strategy to prevent bugs, but i found myself out of time so i opted for this approach)

Monte Carlo modified

The first implementation of the *Monte Carlo* strategy was the original *Monte Carlo Tree Search* (MCTS), meaning that at each iteration a new node was selected between the unexplored ones, expanded and evaluated using random simulation until the end. The strategy used to choose the path for each iteration was to compute:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

This function evaluates the goodness of a node, but also gives importance to nodes that were less considered in previous iterations.

The problem I encountered using this base approach was that it didn't reach enought depth. For each state 44 moves can be performed (excluding those where the tile is occupied by the opponent), so the number of nodes that the algorythm has to consider is something less than $44^{(n\_moves)}$.

Obviously, some moves will end up in already seen states, but the depth this version of the algorythm can reach is more or less ln(n_iter) / ln(44) (at least at the beginning, then the number of new states should decrease). This resulted in the first moves being made consiously and the rest being random moves. On average with 1k iterations 10k states had values in the dictionary. The problem is that these numbers doesn't even compare to the $83x10^9$ possible states of Quixo.

Decreasing eps in the selection function improved the depth reached and the number of seen states, but not enought to produce good results.

I tried also using simmetries, but the problem was still there.

Then i tried to implement it in a different way, more similar to what we saw in the labs, by doing random plays and updating the path based on the outcome. It worked better but still the number of states visited was not enough.

When i was working on the simulation function, I had the idea of using a net to extimate the value returned by the simulation function from the current board that i used later.

I left the *Monte Carlo* approach due to the number of states and the amount of time needed to learn a meaningful table. On average the results with relatively small number of iterations was around 60% winning rate for player_0 and 52% for *player_1*.

Minimax modified

The first implementation of Minimax worked pretty well, obtaining around *93%* winrate with *max_depth = 2*. However, I noticed that increasing the depth decreased the winrate. My opinion is that the random behaviour of an opponent doesn't help with *Minimax*, which imply that the opponent will choose the optimal move.

I tried three different approaches to modify the Minimax strategy:

Minimax_avg

I tried to average the min layer of the *Minimax* in order to mitigate the fact that the opponent play randomly by giving the same value to each possibility. This actually improved the result, obtaining *97%* winrate with *max_depth = 2* and similar results onwards.

Minimax_eval

Here I tried change the evaluation function by adding a bonus for the number of tiles in a line. For each winning line i removed the number of tiles of the opponent from the number of tiles of the player and then raised by power *2* so that the difference of tile in each line was enhanced. The results were almost perfect, with a winrate of 99-100%.

Since max_depth = 0 obtained almost the same results, i created the *Scripted* solution in a different file, just to highlight the fact that this evaluation seems to be enought to almost always beat a random player.

Minimax_weights

Having seen the effect of averaging the min layer, this time I tried adding a weight to the evaluation of each move from the enemy, hoping it would result in a different evaluation for moves that are used at different rates . The weights are trained in an evolutionary way. The results are good even with little training, around *97%* winrate with *Max_epoch = 10*, *Max_population = 10* and *N_trials = 20.*

Minimax itself is a good strategy and it works. It also made me reflect on the potential of an evaluation function.

EA_eval_policy

Having found that the evaluation function was beneficial to the *Minimax* approach, i tried assigning weights to each tile on the board to establish the importance of each tile in the evaluation, subtracting the evaluation for player_1 from the evaluation for player_0. I took the Scripted method and added the weights and the evolutionary algorythm used in *Minimax_weights. The results were around 98% winrate* with *Max_epoch = 10*, *Max_population = 20* and *N_trials = 100.*

Then I tried to modify it a bit the method:

- ***More**:* I took the evaluation function that worked for Minimax, assigned a weight to each evaluation and computed the overall bonus, i was trying to give more importance to specific lines (now that I'm writing the report i noticed that i made an error in the implementation and gave the same weight to all horizontal lines)

- ***Momentum***: I added a weight multiplied by the number of moves to each existing weight trying to obtain different evaluations for different game steps

The results didn't improve if not for the *Momentum+More* which reached 99%

Simulation_conv

When i was working on Monte Carlo i had thought about using a net to extimate the value returned by the simulation function, which tested each state with *n* random plays and returned *win_0 – win_1*. I used the state (which includes the player who has to move) as input. Inside the net the input is passed as it is to a dense layer and in parallel as a board to a convolutional layer, the results of the two are concatenated and passed to a dense layer for the output. A buffer memory is implemented to save *(state, simulation_value),* then batches of data are taken at random but balancing the number of positive and negative values.

The results seemed good enought, around 90% winrate but only if trained for longer than the other models.

Final Considerations

Overall, i tried to avoid running very complex models or long training sessions, on one hand i hadn't much time and on the other hand the professor seemed to prefer thought over power.

WR (%)	MC	Minimax				EA_eval_policy	simulation_conv
		base	avg	eval	weights		
player_0	60	93	98	100	97	98	90
player_1	52	93	98	99	98	97	98

I didn't quite understand if i have to choose a single model over the others, but if i have to, then i choose Minimax_eval.

I was thinking about waiting for the next turn to present the project. I liked doing this project and while I was starting to write the report i noticed that there where still possibilities I wanted to try:

- DQN (which should work for the problem at hand)
- Playing more around EA_eval_policy and on the concept of momentum, also correct my error on More
- Perform some ablations on the current implementation of Simulation_conv in order to check the importance and the correctness of each part
- Giving the inverted board to the net instead of the state or adding the simmetries to the batches
- Using the MCTS strategy but starting from winning positions and going back (i don't know if it's feasible. In theory only the tiles from the last player that played can be used and the tile that enter in the opposite space will be either empty of the same as the player)

In conclusion I'd like to add that this was a very interesting course and I would have liked to follow it live. I hope that the project is good enought and that the report is what you expected.

Thank you for your time