SORBONNE UNIVERSITÉ

Report 2I013
Two Player Game Othello and Awele

LAZAROSKA Marija
SOTIROSKI Filip

May 2019
Paris, France

Introduction

We are given a task to create two board games that can be played by two players. We are interested in coding the structure of the game and create different types of players who can interact with the game structure.
Once we have done this initial phase, our goal is to make players who are able to see a few steps ahead what could be the best possible move that the player can take.
The final part is to make the same players able to simulate games and learn from the good or bad decisions that they provoked during the game

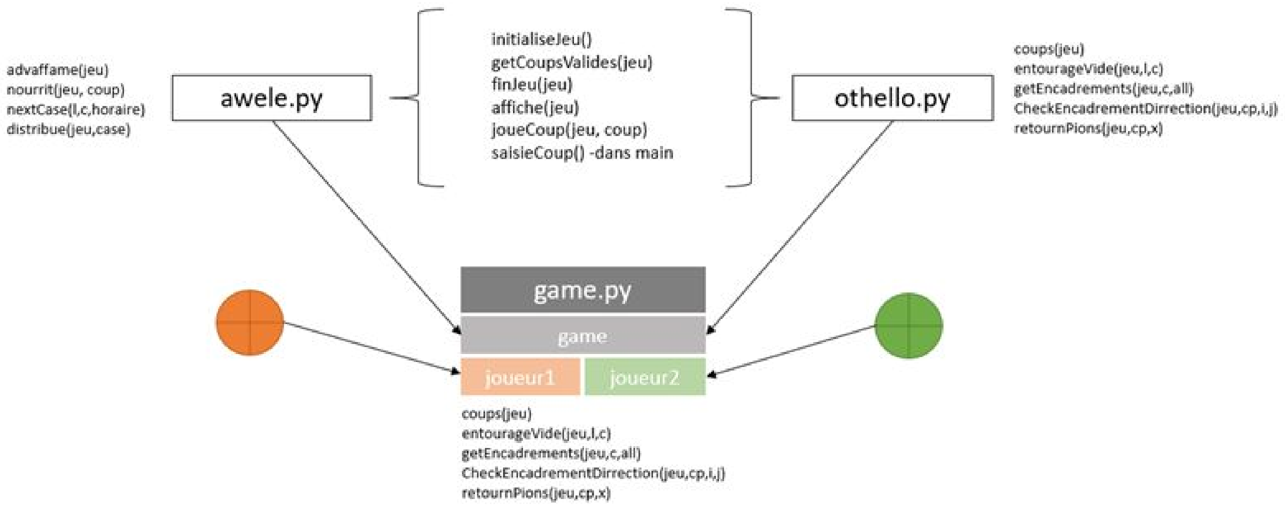
Solution

In order to create a structure that can be used for both games and build the rest of the game in a modular concept we have created the game.py module.
This way, we can have different modules for every player that can interact with the game structure.

The two player games Othello and Awele can be represented in a form of a decision tree where every node is a representation of the board in a different state. We wrote a few heuristic functions for evaluation of the states. Using the search algorithms Minimax and Alpha Beta Pruning our players can decide what is the best move for playing based on the value returned from the heuristic functions.

The training of a player is based on the efficiency of their coefficients for their heuristic functions. We used two ways of training. The first one is local supervised learning where the player was improving himself by playing and randomly modifying his coefficients versus a better player. The second is supervised learning where the player is playing against a better player, and his moves are based and coefficients modified according to the moves from an external ‘expert player’.

Game Structure



1.1 game.py module

game.py

The functions from game.py are used in both games. As Othello and Awele have different functionalities and rules, some of the functions need to be adapted to both games.
For example, there is a different condition for ending Othello and Awele. The function finJeu is being called from the module game.py, but the code is different for both games in their proper modules othello.py and awele.py.
Other similar examples are saisieCoup, getCoupsValides, joueCoup and affiche which are independant for both games.
On the side we have functions that change certain properties in the game. Such as changeJoueur, getScore, getJoueur etc. They are declared in the game.py module.
We also use the variables game, joueur1 and joueur2. In the variable game we assign the module awele or othello that we want to play. For joueur1 and joueur2, we assign the modules of the players that play the game.

othello.py

This game is played on a 8x8 board where players take turns placing disks. During a play, any disks of the opponent's color that are in a straight line and bounded by the newly placed disk and another disk of the current player's color are turned over in the current player's color.

The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

In the othello.py module we have coded the following functions that manipulate with the game structure. The game is initialised with 2 black and 2 white disks in the middle of the board.

The *coups(jeu)* function returns us a list of possible moves around enemy's disks. We also call the function *entourageVide(jeu,l,c)* that checks if there is a entourage in a certain case on the board.
returnPions(jeu,cp,x) is a function that swaps the disks when a move has been effected.
checkEncadrementDirection(jeu,cp,i,j) is a function that looks in all directions from a certain position and tells if there is a framing(encadrement) in that direction. The function is then used in *getEncadrements(jeu,c,all)*.
The game ends *finJeu(jeu)* when there are no moves available, the board is full or there are only one color disks.

awele.py

The game requires a board with two straight rows of six pits. Initially, 48 seeds are used, 6 in each pit. The game is being played anti-clockwise. Player one has the first row on the top, player two the bottom. The game ends *finJeu(jeu)* when one player has more than half (24) of the score. Each player picks a pit of seeds and inserts sequentially in the next pit in an anti-clockwise direction. They score when the last seed is put in a pit with one or two seeds, and they are picked with the previous one's if there is a chain of 2 and 3 seeds in the previous pits.

The *distribue(jeu,case)* function uses the nextCase function to spread the seeds from one case.
nextCase(jeu) is used to pass from one row to another when we reach the end of the row.
advaffame(jeu) makes sure that the opponent doesn't starve so that he has a next move to make. It is used in the *nourrit(jeu,coup)* function to check if the seeds from a certain move/pit can reach the opponent's row.

main.py

In this module, we find the principal while loop in which players take turns using the function saisieCoup(jeu) that returns a move to be played.
First of all, we assign the game that we play and the players who play this game by using the variables game, jouer1, joueur2 from the game.py module.
After the game is initialised, each player is allowed to take 100 turns. If this condition is fulfilled or if the finJeu(jeu) returns true, then the game finishes and displays the final results.

To make the interaction with the game more intuitive for humans, both games have different affiche(jeu) functions that print the state of the board in the console.

main_permutation.py

The principle is almost the same as in main.py. But here, we can set how many parties we want to be played. The only small difference is that the first 4 turns of the game are played randomly by the random player in order to avoid deterministic parties. There is no graphical illustration of the game, but only the stats for games one by each player are printed in the console.
It is also important to switch the order of the players, as some players perform better when playing as first or second.

Joueurs

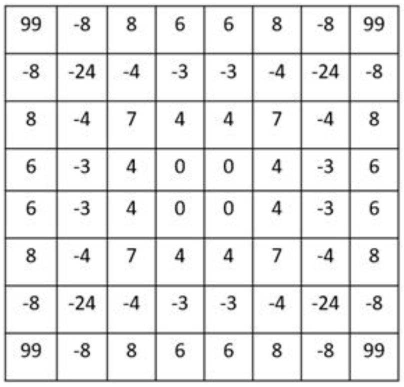
joueur_humain.py
The human player inserts the position of the possible move, and if it is valid, the move is played
joueur_random.py
The random player chooses a random move from the function getCoupsValides(jeu)
joueur_premier_coup.py
This player chooses the first possible move on the board.
joueur_horizon.py
A player who finds his best move that gives him maximal value from the evaluation function while playing his
joueur_minimax.py
A player who uses the Minimax search algorithm.
joueur_alphabeta.py
A player who uses the Alpha Beta pruning algorithm.

Heuristic functions

The heuristic functions are used in the evaluation of the board. Because every node represents a different state of the board, the same state can be evaluated by these functions. The functions return a different value according to the evaluation of the state.
This is a key part for making a strong player as the evaluation functions represent certain strategies that can be used in the game. The only downside is that they need to be multiplied by certain coefficients whose values are given by us in the beginning, but can be improved later with the training.

Strategies/evaluation functions used in Othello:

- **Corner evaluation**
  The corners are the most important positions on the board. Once there is a disk in the corner, it cannot be flipped and can flip diagonals and wall lines.
- **X, C positions evaluation**
  The positions X and C are the positions that are around the corner. In case we have a disk on those positions, the corners are opened for the opponent to place his disk.
- **Positional evaluation**
  A list of lists is used to store the likelihood that a certain position is better than the other one. Therefore, we sum the values of the positions where our disks are placed.



99	-8	8	6	6	8	-8	99
-8	-24	-4	-3	-3	-4	-24	-8
8	-4	7	4	4	7	-4	8
6	-3	4	0	0	4	-3	6
6	-3	4	0	0	4	-3	6
8	-4	7	4	4	7	-4	8
-8	-24	-4	-3	-3	-4	-24	-8
99	-8	8	6	6	8	-8	99

1.2 positional values(Othello)

- **Score evaluation (difference)**
  The difference in score is important if we want to win this game. But, aiming on having a bigger score can be countered in the last few turns in the game, as having bigger score doesn’t necessarily mean winning the game.

Strategies/evaluation functions used in Awele:

- **Score evaluation (difference)**
  The first player to have a score over 24 wins the game, so it is better to have higher score difference against the opponent.
- **Difference in seeds on the board**
  It’s important for the current player, after he played his move, to have more seeds in his side of the board than on the opponent's. Therefore this evaluation calculates the difference between the number of current player’s seeds and the opponent’s. The move

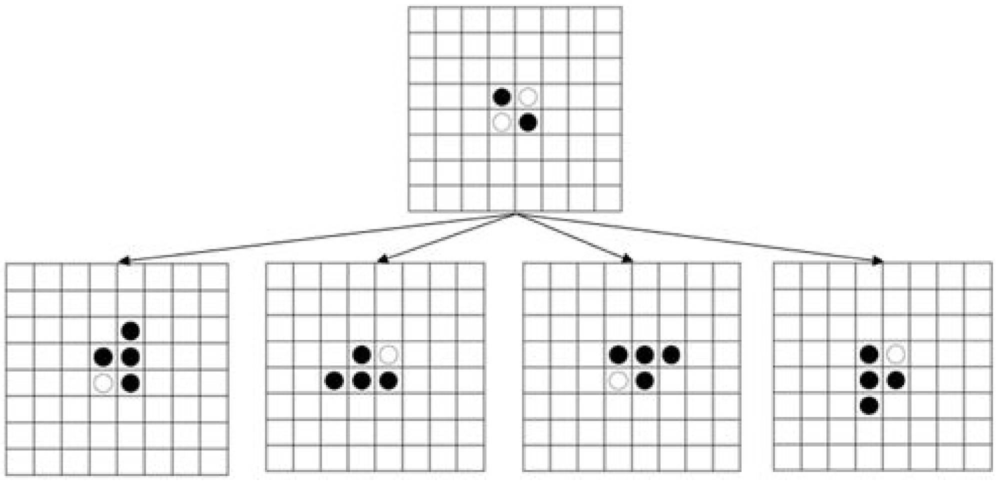
is better if the difference is bigger. This creates a bigger probability to have pits with one or two seeds on the opponent's side.

- **Line evaluation**
  To avoid losing, the current player should avoid playing moves which empty his holes leaving only 1 or 2 seeds in them**.** The less there are holes with 1 or 2 seeds on the current player's side, the move is better.

- **Opponent's line evaluation**
  The opposite of the line evaluation, it is better for the current player to leave holes on the opponent's side with 1 or 2 seeds.

Tree search

The three most import functions in the process of decision making and choosing the best move to make, implemented in search algorithms such as MiniMax and Alpha Beta pruning, are evaluation, estimation and decision functions. As both of the games can be represented as a tree were every node represents a state of the game, we use those functions to search in the tree starting from the root (state before playing a move) to a given depth n (the state after n consecutive moves) and return the best value. The functions have the same structure for both games, only the heuristic functions that are used differ.



1.3 Search tree(depth 1)

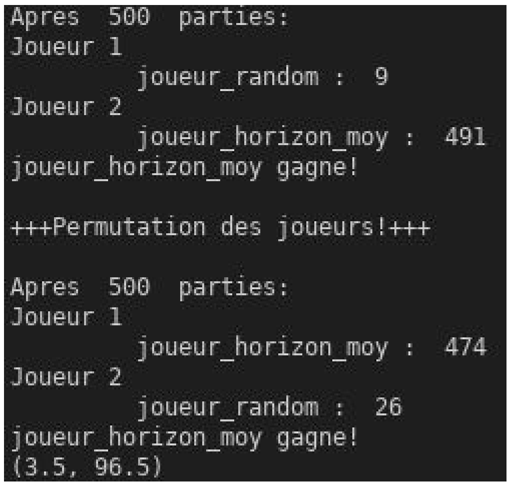
Evaluation
The evaluation function *evaluation(jeu)* returns the scalar product of the returned values of the heuristic functions and the coefficients for each heuristic function. At the beginning this coefficients are given instinctively so there is no guarantee that their values are the best choice, meaning other coefficients might lead to choosing a better move. The best value for them is chosen later with the training.

Estimation
Estimation function is a recursive function which is being used in the search algorithms to return the value corresponding to the best leaf node. It searches until a given depth N. There are two cases for ending the search until the N node. One of them is in case the game ends before reaching depth N. The second one is when reaching depth N. In this case the evaluation function is being called for each leaf node.

Decision
The decision function returns the best move of the estimation function. For each possible move, a certain value from the estimation function is returned. Depending on the best value, the best move is being chosen.

Horizon

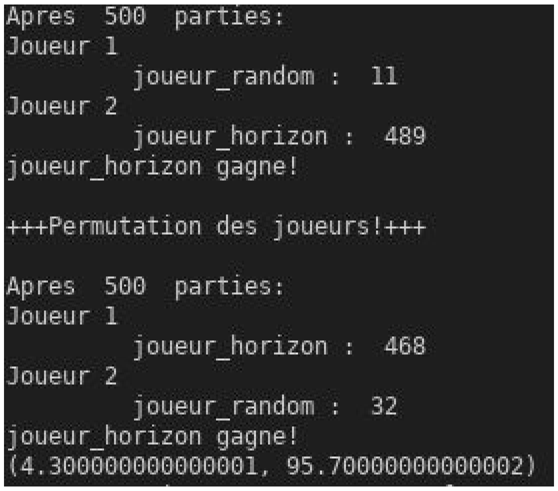
The Horizon 1 player uses the three functions decision, estimation, evaluation in order to choose the best move from depth 1. It takes every possible move, plays that move and evaluates the situation of the board.
If the depth is bigger than 1 (Horizon N), it plays the consecutive moves for each previously played move and then it evaluates the board recursively by using the function estimation.
If we play against a random player we can use the strategy to return the average value when the random player needs to choose his move, and when it's our turn we play the move with the biggest value.
To do this we created a joueur_horizon_moy player who goes to depth 2 where he returns the value in the list closest to the average and on depth 1 the biggest value.
The only function for evaluation that we chose was the one that looks the score.
We notice that after 1000 parties, the Horizon Average wins 96.5% of the parties and the Horizon 1 wins 95.7%. The results are close but if we do the exact same thing with the average strategy on a deeper depth by using the AlphaBeta algorithm, the results would be better for the Horizon Average.



```
Apres  500  parties:
Joueur 1
        joueur_random :  9
Joueur 2
        joueur_horizon_moy :  491
joueur_horizon_moy gagne!

+++Permutation des joueurs!+++

Apres  500  parties:
Joueur 1
        joueur_horizon_moy :  474
Joueur 2
        joueur_random :  26
joueur_horizon_moy gagne!
(3.5, 96.5)
```



```
Apres  500  parties:
Joueur 1
        joueur_random :  11
Joueur 2
        joueur_horizon :  489
joueur_horizon gagne!

+++Permutation des joueurs!+++

Apres  500  parties:
Joueur 1
        joueur_horizon :  468
Joueur 2
        joueur_random :  32
joueur_horizon gagne!
(4.300000000000001, 95.70000000000002)
```

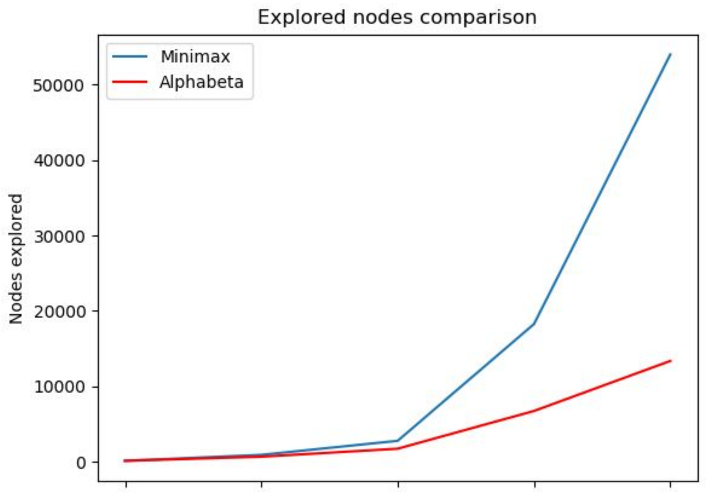
Minimax

The philosophy of the minimax algorithm is to maximize the minimum score. We suppose that we want to play the best move, and the opponent chooses the move that arrange us the least while using the same strategy as us. Therefore, we want to maximize the value on our turn and minimize when it is the opponent's turn.
The function estimationMin() is used on the nodes that the opponent chooses, and estimationMax() on our nodes when it is our turn. They are both being recursively called one in another.
In the decision function we are calling the estimationMin() function, and we choose the move with the maximal value.

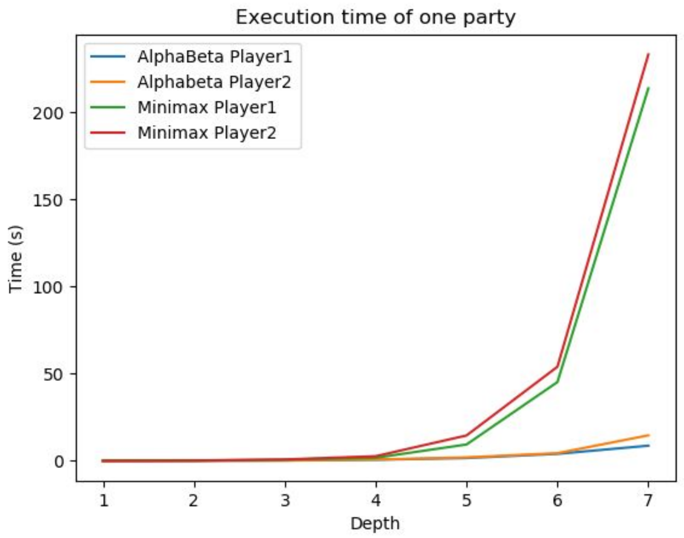
To maximize our chances of winning. We implemented a MaxAvg player (joueur_minimax_avg) if the opponent is a random player. The functioning is the same as we mentioned in the Horizon paragraph when we created the joueur_horizon_moy. But this time, it is implemented with the minimax algorithm.

Alpha Beta pruning

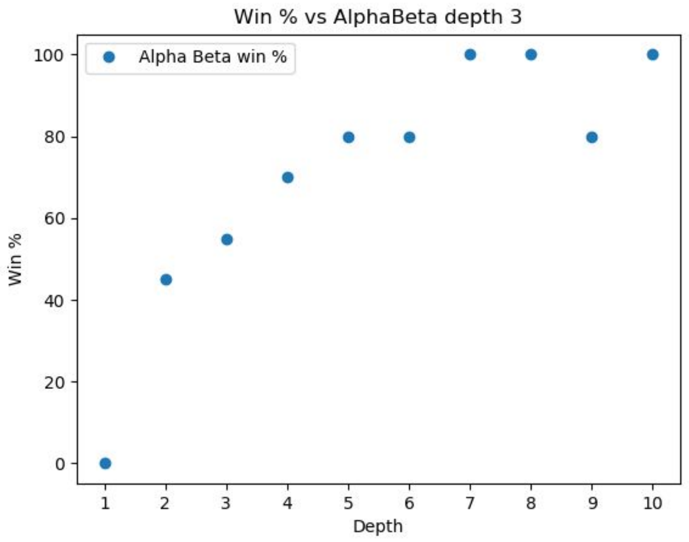
The Alpha Beta pruning algorithm is similar to the Minimax algorithm but explores less nodes than Minimax, excluding nodes that don't need to be evaluated. The algorithm uses upper and lower bounds, alpha and beta in the process of excluding unimportant nodes. It stops evaluating the node when one of the node values is out of the bounds, which proves that further branches should not be evaluated.
The Alpha Beta algorithm gives exactly the same result as Minimax, but in a shorter time, omitting some branches.



If we compare the explored nodes from both Minimax and AlphaBeta players playing one against each other on the same depth we will see that from depth 4, the AlphaBeta algorithm explores much less nodes compared with Minimax. For this plot, we played total of 10 games for each depth.
Each player played 5 games as player 1 and 5 as player 2.



Here we show the execution time of every algorithm. We show 10 games of the Alpha Beta and Minimax algorithm versus premier_coup_valide playing as player one and two.
We notice that until depth 4, the execution times of both algorithms are almost the same. As the depth gets deeper, the algorithm needs more time to choose the best move. We also see that for both algorithms when the player plays the second turn as player 2, needs more time to finish the game compared to the time playing as player 1.



In this plot, we play 10 parties with our Alpha Beta player on different depth against a static Alpha Beta player on depth 3 with the same functions.
We want to show that the bigger depth does not guarantee that the player will always win. Another problem is that as the depth becomes bigger, the game uses more resources such as memory and time.
Therefore for optimal performance, it is good to choose the depth between 4 and 6.

Apprentissage

We previously mentioned the issue that we have when choosing the coefficients for the evaluation functions.
Now we want to create a training for our player in a way that tests different parameters and modifies the values. Therefore, we use two types of learning, weak and supervised.

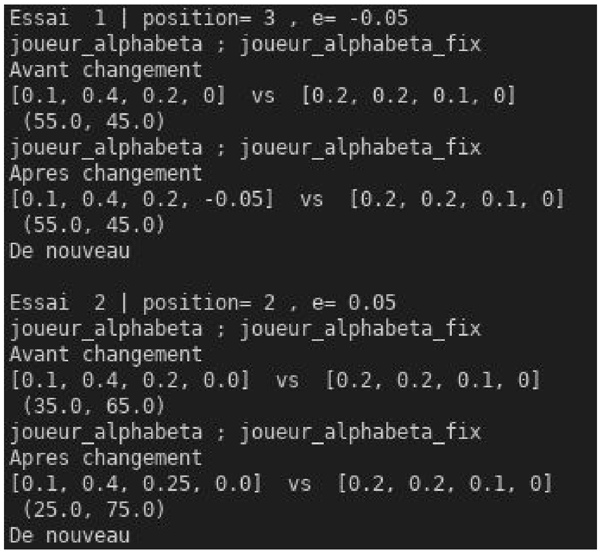
Local Supervised Learning

The objective of this learning is to improve the parameters while playing against a better opponent. We first simulate N games with our old coefficients against the opponent, and N games with the new changed coefficients. If we see improvement in the games won, the changed coefficients are accepted by the player. If the result is inferior than the one with the old coefficients, the process is repeated but new coefficients are chosen again.
The way we change the coefficients is that a random coefficient is being chosen to be modified by increasing or decreasing its value for epsilon (e).

There are several issues with this kind of learning.

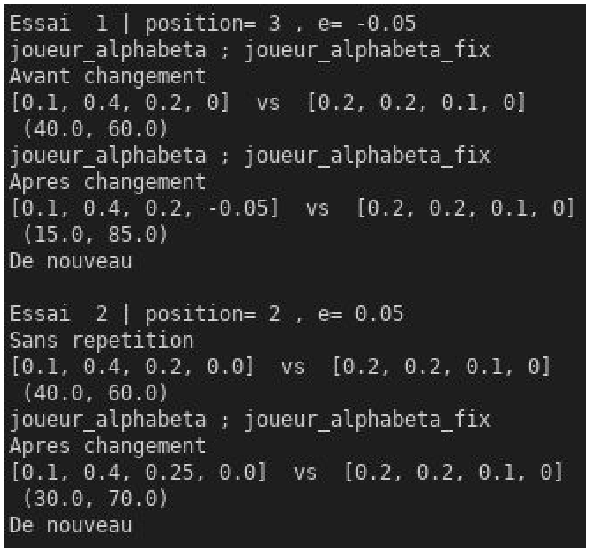
The first problem is the different win percentage when the player simulates games before he changes his coefficients. This happens mostly because we are playing a small amount of parties, and the results can vary. This only occurs when the old coefficients are better than the new ones. We notice that it takes a lot more time for the learning to be made as we are playing the same game with same coefficients over and over. In order to resolve this problem we introduced the function called *apprentissageFaibleSansRepetition*
But still, by using this one we won't get the best results, but we will save time. For example:

On the left picture we see the learning done by the function that does not avoid the repetition problem. We see that when playing for the first time against the same coefficients we get first 55% win, and later 35% win.
The one on the right uses the previous result in ‘Essai 1’ and does not replay the same game again. Therefore, if the player didn’t learn, it will use the same result of 40% and try to get a better score with the new parameters.



```
Essai  1 | position= 3 , e= -0.05
joueur_alphabeta ; joueur_alphabeta_fix
Avant changement
[0.1, 0.4, 0.2, 0]  vs  [0.2, 0.2, 0.1, 0]
 (55.0, 45.0)
joueur_alphabeta ; joueur_alphabeta_fix
Apres changement
[0.1, 0.4, 0.2, -0.05]  vs  [0.2, 0.2, 0.1, 0]
 (55.0, 45.0)
De nouveau

Essai  2 | position= 2 , e= 0.05
joueur_alphabeta ; joueur_alphabeta_fix
Avant changement
[0.1, 0.4, 0.2, 0.0]  vs  [0.2, 0.2, 0.1, 0]
 (35.0, 65.0)
joueur_alphabeta ; joueur_alphabeta_fix
Apres changement
[0.1, 0.4, 0.25, 0.0]  vs  [0.2, 0.2, 0.1, 0]
 (25.0, 75.0)
De nouveau
```



```
Essai  1 | position= 3 , e= -0.05
joueur_alphabeta ; joueur_alphabeta_fix
Avant changement
[0.1, 0.4, 0.2, 0]  vs  [0.2, 0.2, 0.1, 0]
 (40.0, 60.0)
joueur_alphabeta ; joueur_alphabeta_fix
Apres changement
[0.1, 0.4, 0.2, -0.05]  vs  [0.2, 0.2, 0.1, 0]
 (15.0, 85.0)
De nouveau

Essai  2 | position= 2 , e= 0.05
Sans repetition
[0.1, 0.4, 0.2, 0.0]  vs  [0.2, 0.2, 0.1, 0]
 (40.0, 60.0)
joueur_alphabeta ; joueur_alphabeta_fix
Apres changement
[0.1, 0.4, 0.25, 0.0]  vs  [0.2, 0.2, 0.1, 0]
 (30.0, 70.0)
De nouveau
```

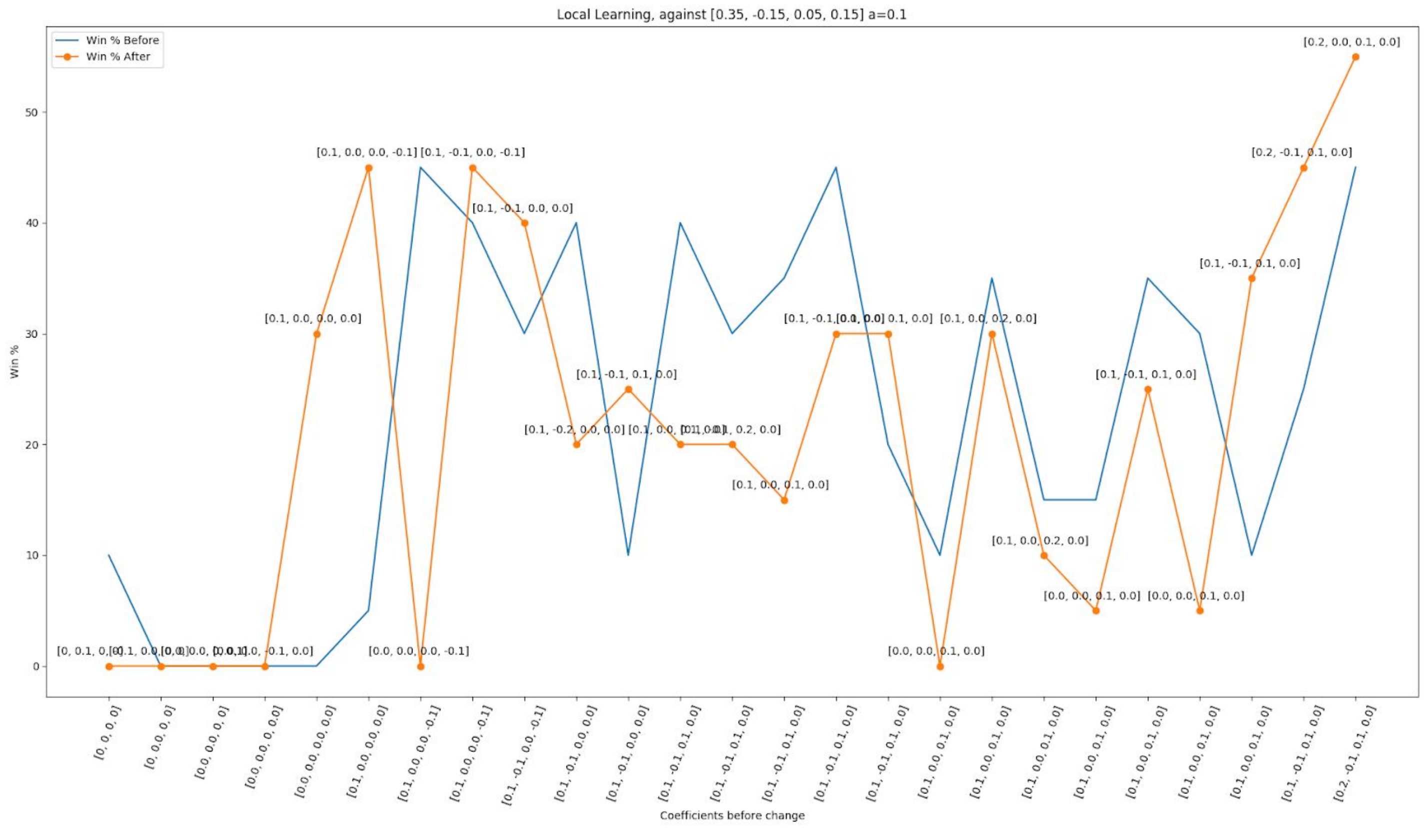
The second more difficult problem is that our coefficients can’t improve our score because of epsilon’s value. In fact, there is a big probability that the set of parameters will tend to the same values, therefore we will only reach the “local best coefficients”. In order to avoid this, the epsilon value should become bigger so that we can explore other coefficients and maybe get better results.

Test

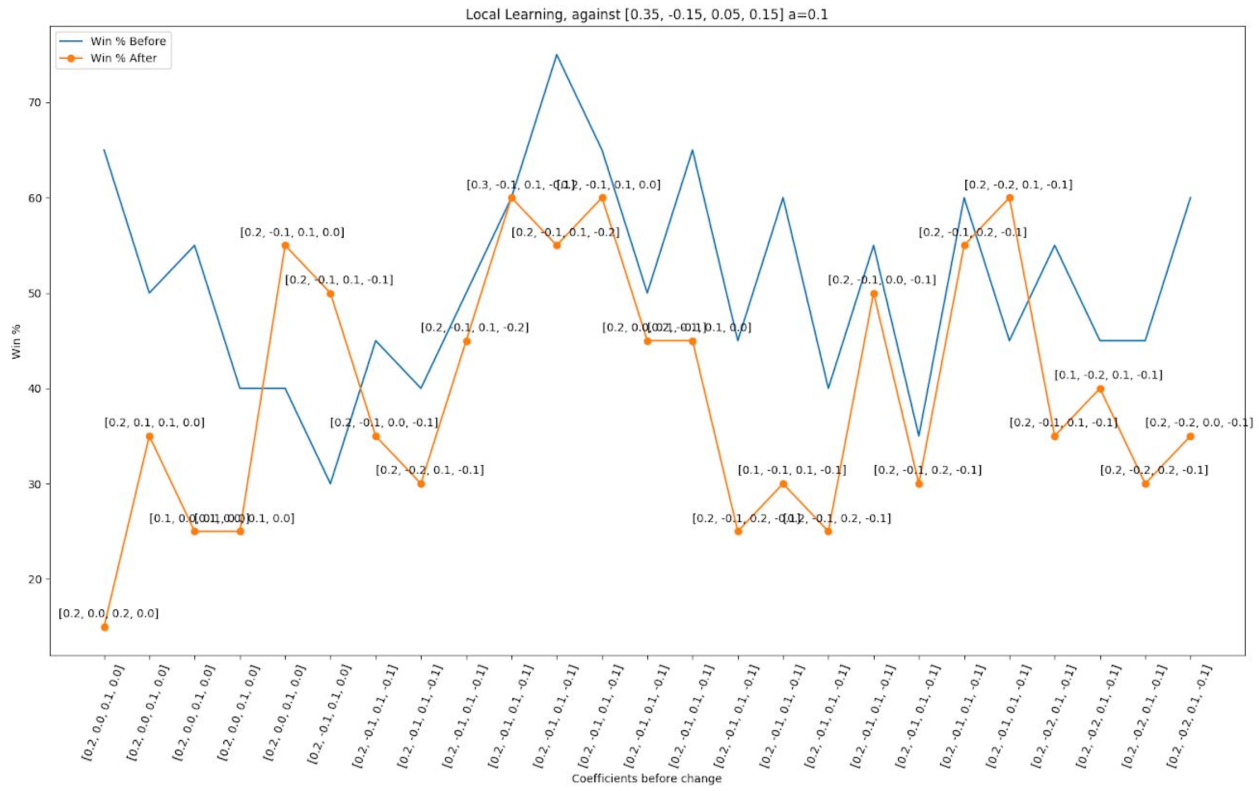
We are setting our player with all null coefficients. A random coefficient is chosen and it's value is added/reduced by epsilon. If after a few parties we get better results with the changed coefficients, these are applied to the player. If not, another coefficient is randomly changed.

We suppose that the alpha beta player with the coefficients [0.35, -0.15, 0.05, 0.15] is a strong player. We set our student to have coefficients [0,0,0,0]. The blue line indicates the win percentage when playing with the coefficients in the xaxis (before the change), and the orange one is after the change. If the orange has a better result than the blue one, the parameters are changed. The annotation on the data points is the new coefficient

The last coefficient that has a superior win percentage is considered as the best coefficient and will win against the opponent. The number of games that were played were 20, 10 as player 1, 10 as player 2.



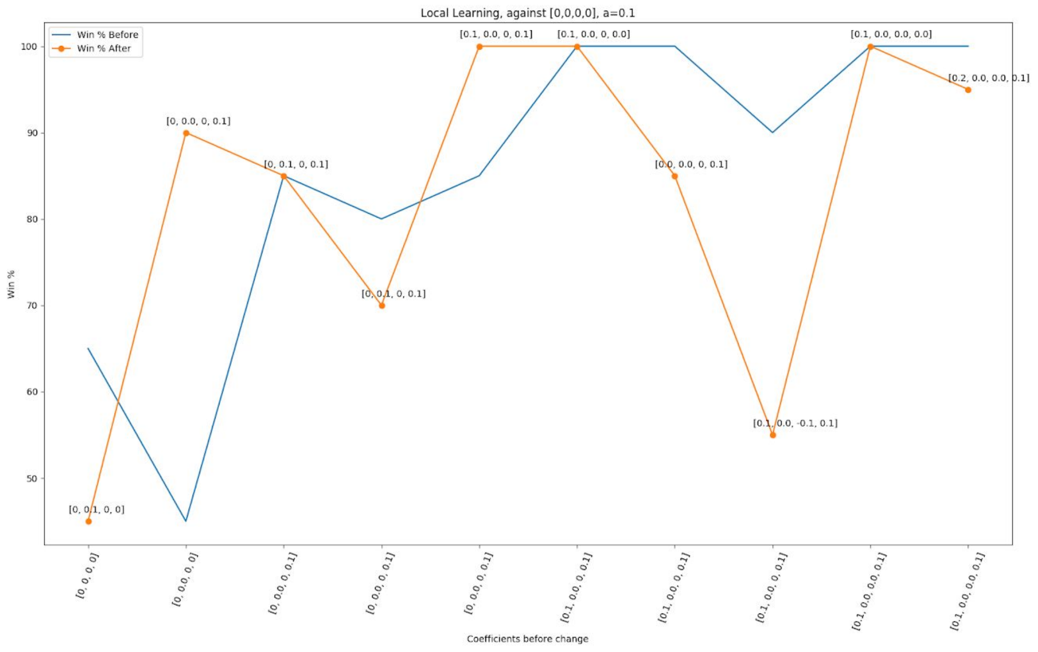
local1.png



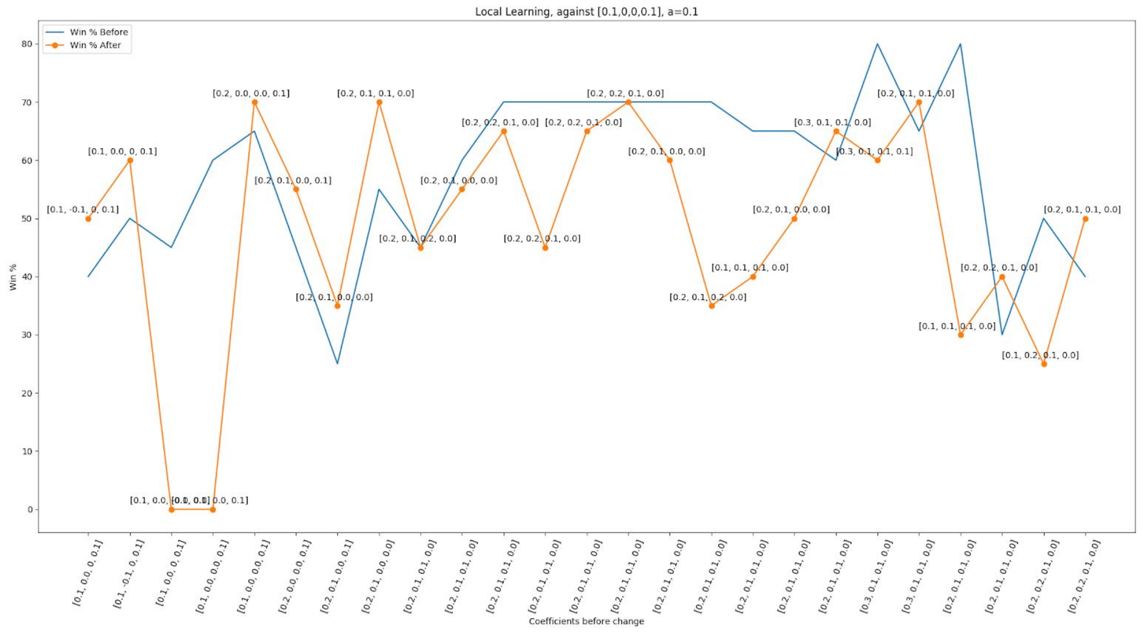
Local Learning, against [0.35, -0.15, 0.05, 0.15] a=0.1
Win % Before
Win % After
Win %
Coefficients before change

local2.png

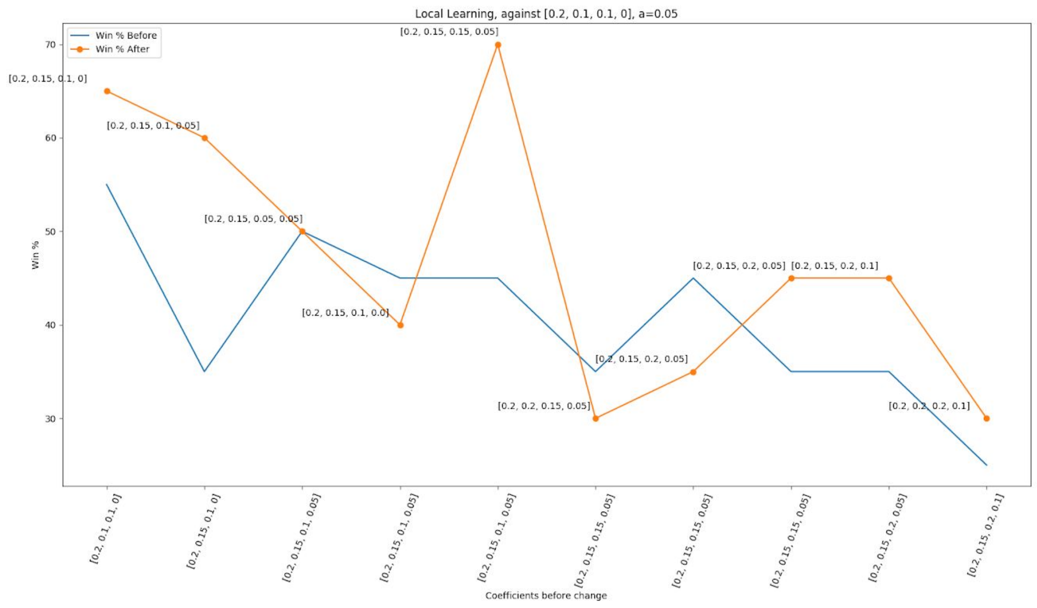
If we want to make our player learn by himself, we can make him play with his null coefficients against an opponent with null coefficients. After a few parties we can notice that the student's score is better than the opponent's. Then we set the student to play against his current best coefficient. In our third try, we decrease the epsilon value to 0.05 to have more precise and better coefficients.



local3.png



local4.png

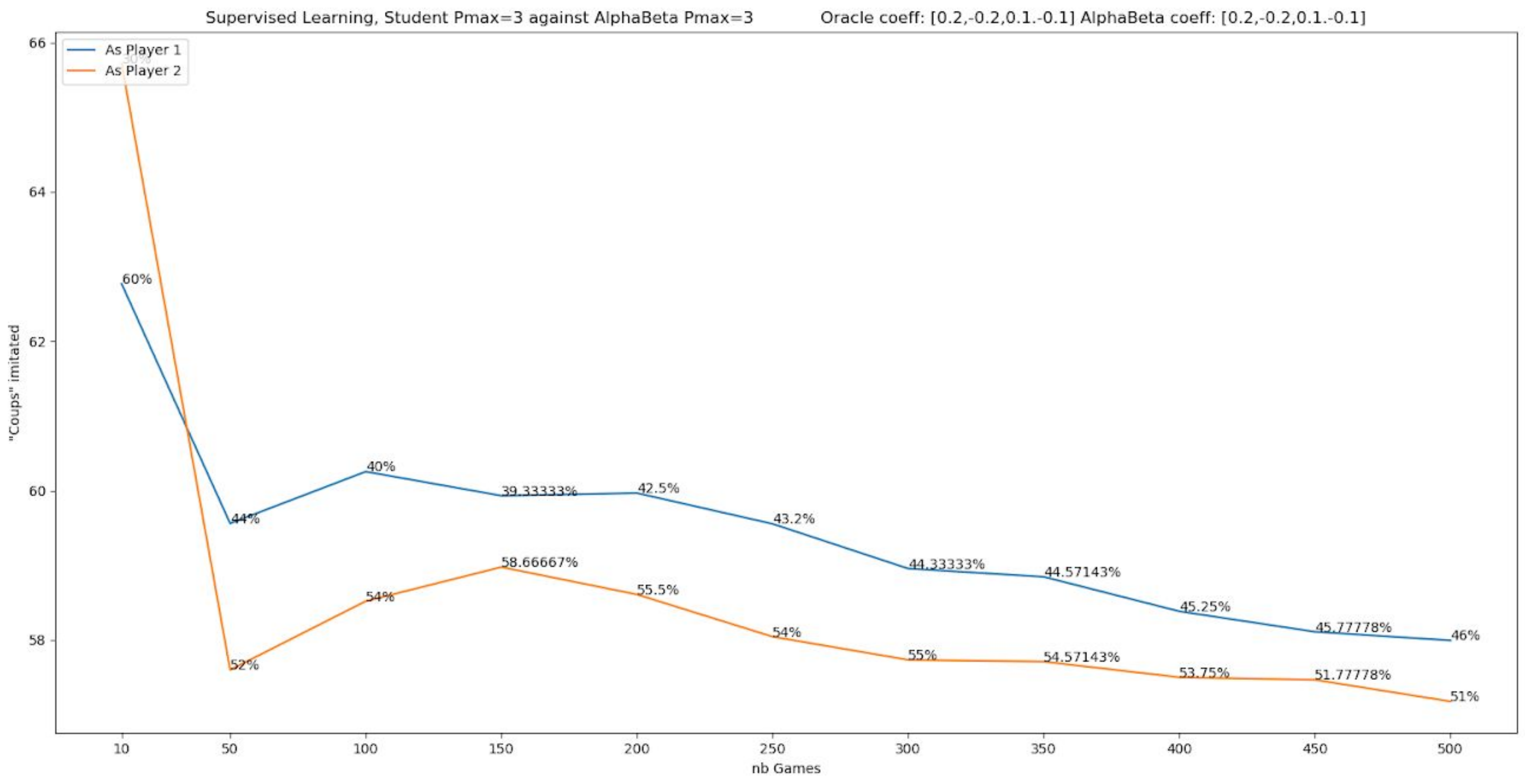

Local Learning, against [0.2, 0.1, 0.1, 0], a=0.05
Win % Before
Win % After
Win %
70
60
50
40
30
[0.2, 0.15, 0.15, 0.05]
[0.2, 0.15, 0.1, 0]
[0.2, 0.15, 0.1, 0.05]
[0.2, 0.15, 0.05, 0.05]
[0.2, 0.15, 0.1, 0.0]
[0.2, 0.15, 0.2, 0.05]
[0.2, 0.15, 0.2, 0.1]
[0.2, 0.15, 0.2, 0.05]
[0.2, 0.2, 0.15, 0.05]
[0.2, 0.2, 0.2, 0.1]
[0.2, 0.1, 0.1, 0]
[0.2, 0.15, 0.1, 0]
[0.2, 0.15, 0.1, 0.05]
[0.2, 0.15, 0.1, 0.05]
[0.2, 0.15, 0.1, 0.05]
[0.2, 0.15, 0.15, 0.05]
[0.2, 0.15, 0.15, 0.05]
[0.2, 0.15, 0.15, 0.05]
[0.2, 0.15, 0.2, 0.05]
[0.2, 0.15, 0.2, 0.1]
Coefficients before change

local5.png

Supervised learning

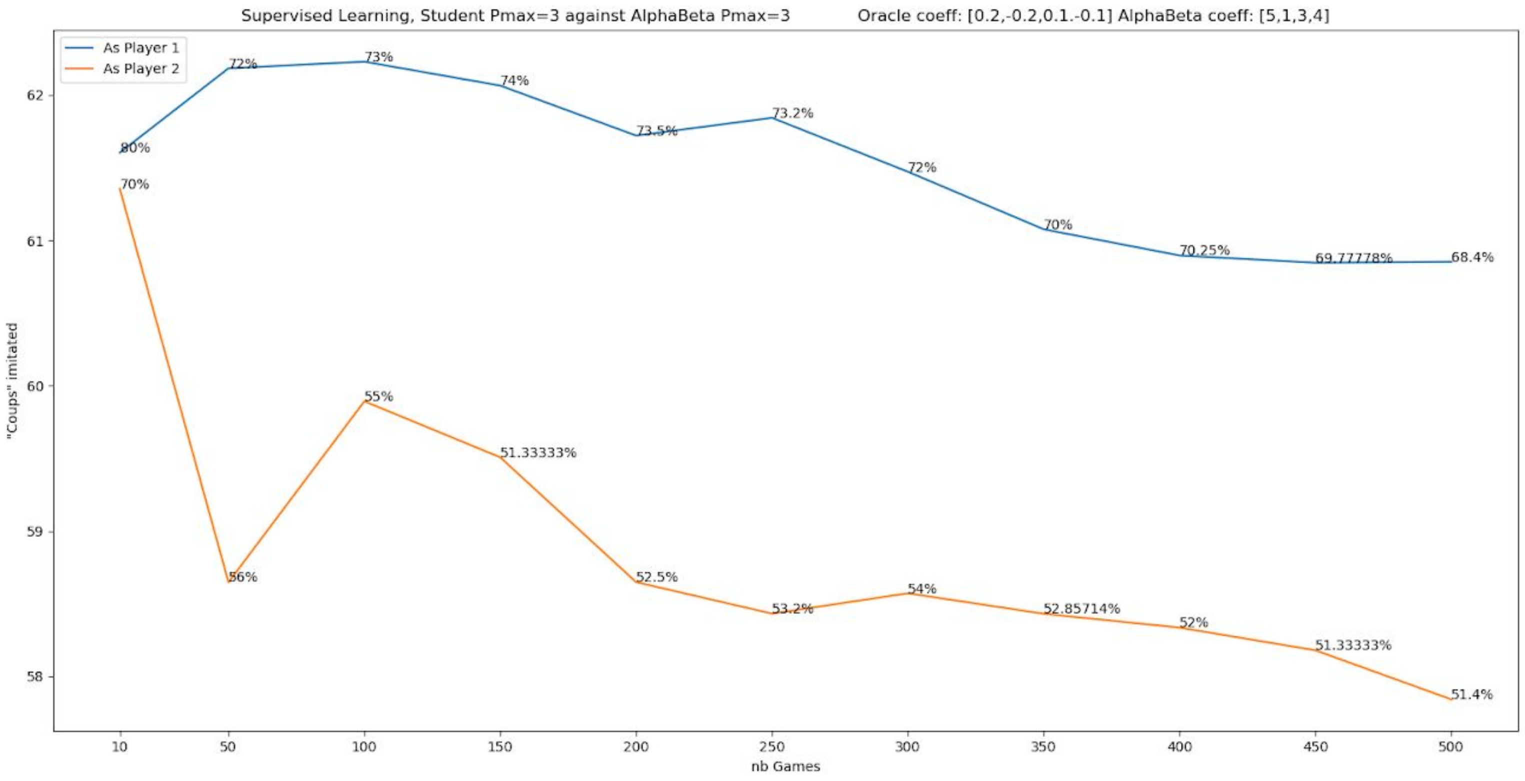
The supervised learning is a more improved training. In this algorithm we have 3 players: the teacher, the student and the opponent. FIrstly we choose the best move that the teacher played. After extracting the best move, the student plays the less good moves that the teacher played. If the score after playing the bad moves is better than after playing the teacher's best move, the coefficients will be modified. We stop the learning when the algorithm achieves an acceptable level of performance. We don't play against the teacher,but we use it as a supervisor of the learning process like a training dataset.

The following graphs are tests from the function apprentissage_supervise for Awele. In our tests we play against AlphaBeta player with different coefficients. The annotation on the data points is the percentage of games won. For the tests our player and opponent have the same depth.



The coefficients we used for Oracle and AlphaBeta are those that were proven the best from the previous tests with the Local learning. Our student plays a game as a first player and adjusts his

coefficients and then plays one more game as a second player so he can adjust his coefficients for when playing as a second player.
In the first graph the results vary, but we can see that the changed coefficients are more suitable for playing as a second player. Also as a first player, the less he imitates the moves of the Oracle the better score he gets, which means that he has improved and doesn’t need to follow his teacher’s moves.



In the second test our player plays against AlphaBeta with [5,1,3,4] as coefficients. The player doesn’t improve himself. The less he imitates the Oracle moves the less he wins.