BERTOMEU Tom
DUFILS Nathan

S1 A1 / Group n°25 - January 13th 2022

REPORT

SAÉ 1.02

1 Presentation of the first AI's strategy :

The first AI plays the first move it can. It can be considered as a « naive » AI.

It browses the game looking for a pawn of its colour. For each one it finds, it determines if it can move, and if it can it plays the first move it can play, otherwise it continues browsing the game until it finds a movement it can play.

Detailed explanation :

The first AI, called "jouePremier", has 2 loops to browse the table representing the game named "*state*". If it crosses an element matching its colour and for which the boolean method *peutBouger()* returns true, it takes its coordinates and makes a list of the positions it can play with the method *possibleDests()* to simulate each position until it finds one that is possible. When it does, it returns a list of two elements, the first one being the source of the movement to do and the second one being the destination of the movement.

2 Presentation of the second AI strategy's :

The second AI plays randomly until a pawn can block on a possible next move in a certain direction. For this AI, it browses the game to search a pawn that can block himself at a possible next move. If it finds a pawn that block himself at a possible next move, it returns the pawn and the destination that blocks him. Otherwise It plays randomly until the first condition will be true.

Detailed explanation :

The second AI called "debutant" was built into two parts where we can find four additional functions :

- At first the functions called "peutJouer" determines if a pawn can move. To make this, we have created a table called "possDest" which contains all its possible destinations. Next we have created a loop that browses the game and tests for every possible destination if the pawn can move. If it does at least one time, this function returns True, otherwise it returns False.

- Secondly the function called "peutBloquer" determines if a player can block himself at a next move. To make this, we have created two loops to browse the table which contains positions of all the pawn in the game and looks if every pawn belonging to the AI can move with the function

“peutJouer”. Then we have created a table called “possDest” that we appended by the possible destinations for the pawn. Using a loop, we browse the possDest table and we test each time if for each possible destination, it can actually play and at the end it returns True if it can play, else it returns False.

- Thirdly the function called “getRandomElement” which takes a list as an argument and returns, using a command that returns a number between 0 and the length of the list, a random element from the list.

- Fourth the function “listePions” which returns a list of the positions of all the pawns of the color of the player. To do this, we create a list of pawns of size 13 because there are 13 red pawns and 13 blue pawns, we also implement a variable that will serve as a counter. Then we have a double loop that browse the table of pawn positions to pick the coordinates of the pawns that belong to the AI using a condition that looks if the location contains pawns of the correct color. If the condition is true, the list of pawns is incremented by the position of the pawn and the counter is incremented by 1. At the end it returns the list of pawns.

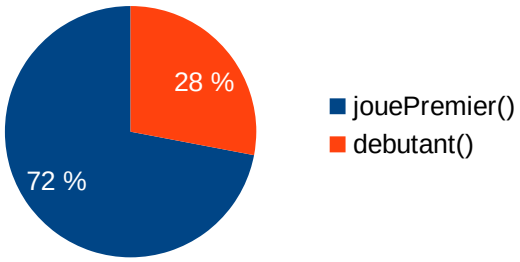
After explaining the additional functions, we can explain the main function **:**

In the first part of this function, we have a condition that looks if a pawn can be blocked in one move thanks to the function “peutBloquer”. If the condition is true, a double loop browse the table of pawn positions seeks the position of a pawn that can be blocked using a condition that uses the function “peutJouer”, if the condition is true we use a table which is incremented by the possible destinations that the pawn can go to and then a loop looks if the possible destination will block the pawn using a condition that uses the function “peutJouer” and if the condition is true, the function returns the pawn source and the destination that blocks the pawn. If the starting condition is not true, we enter the second part of this function where we start by creating a table that listing the pawns that the AI can play and then pass this in the function “getRandomElement” to choose a pawn randomly. Then a "while" loop will see if we can play this random pawn using the function “peutJouer” and if we can't play, we choose another random pawn until we find a playable pawn. After we create a new table where we increment as a value the different possible destinations of the pawn in a “while” loop, we test if a destination chosen at random is functional and as long as it is not we look for another destination at random. Finally we return the source pawn and the destination location.

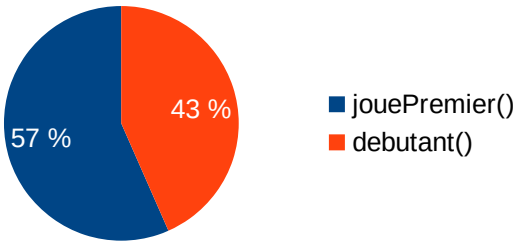
3 Algorithm comparison :

The best way to compare our artificial intelligences is to make them face each other. That's why we've simulated 1 million games by letting *jouePremier()* begin, then another 1 million games by letting *debutant()* begin. The statistics obtained with this process are the following:

When *jouePremier* begins



When *debutant()* begin



We can notice that even tough *debutant()*'s strategy is more advanced than *jouePremier()*'s, it is statistically inferior to it. However, we can deduce an important aspect of the game from this AI: the fact that it plays partly randomly implies that against an opponent whose game will always be the same between each game, the 28 and 43% of victory all constitute sequences of movements where it will be sure to win each time. If the *debutant()* played the first possible move at the first possible position instead of playing randomly when it cannot block, the game would be one-wayed which would not have been interesting to study. During our simulations, for example, we noticed that the main reason why *debutant()* lost was that by trying to block, it reduced the number of moves its opponent had to make to win. It would have been necessary, instead of playing randomly, to play moves that would have increased the number of moves necessary for the opponent to win.

We had many other ideas for such strategies that we could have implemented if we had invested more time in the development of this project. However, we managed to implement an idea, admittedly basic, but producing results that we can use to improve it.