Dept.SUPINFO Academic

Algorithms in Python

First Mini - Project

Three little strategy games

SUMMARY

Version 1.0
Last update: 28/10/2016
Use: Students/Staff
Author: Laurent GODEFROY

1 PRÉAMBLE

This exam is **individual**.

Any form of plagiarism or using codes available online or any other type of support, even partial, is prohibited and will cause a 0, a cheater mention, and even a disciplinary board.

You must return an archive with the format « .zip » containing the source code of your project before **Sunday, November 27 2016 at 23h59, Paris time**. Beyond this date and this hour, you won't be able to send your project and you will get a 0.

There is no viva for this mini-project.

An indicative grading **scale** is given at the end of this subject.

The goal of this mini-project is to program three little combinatory and abstract strategy games in Python.

2 BREAKTROUGH

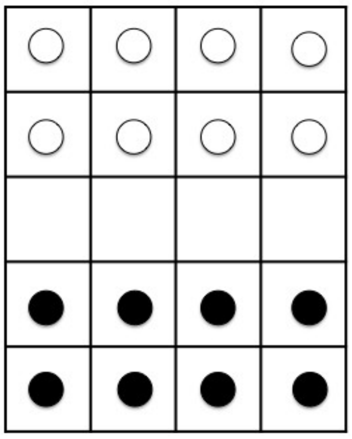
2.1THE RULES OF THIS GAME

Important note: no code is requested in this sub part, where we will explain the rules.

This game has been created in 2000 by Dan Troyka[1]. Two players fight each other on a rectangular board of **n** rows (**n>4**) and **p** columns, the first player has white pawns and the second one has black pawns. At the beginning of the game, the first player's pawns are on the two first rows of the board and those of the second player are on the two last rows. The others cases are empty.

1http://mancala.wikia.com/wiki/William_Daniel_Troyka

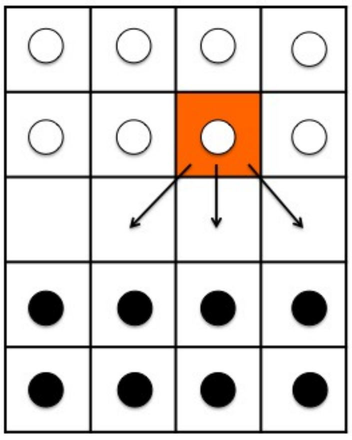
Here is an example of the initial configuration with a board of 5 rows and 4 columns:



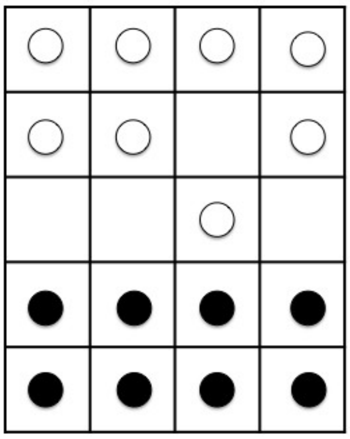
White begins, then players will move one of their pawns in turn, respecting the different following constraints:

- We must move forward, i.e. White pawns must go down and black pawns must go up. We cannot stay on the same row.
- A pawn can move to an empty case if the case is adjacent orthogonally or diagonally relative to its current position.
- A pawn can move to a square occupied by a pawn of the other player only if this square is adjacent diagonally relative to its current position. It then « take » the pawn.
- The taking of a pawn is not priority in relation to a movement to an empty square. We can not do multiple takes during one turn.
- A pawn can never move to a square that contain a pawn of the same colour.

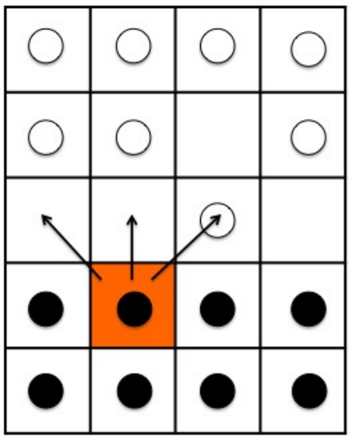
Here is an illustration to explain these constraints. From the initial configuration, the white pawn located on the second row and on the third column can move on one of these three squares:



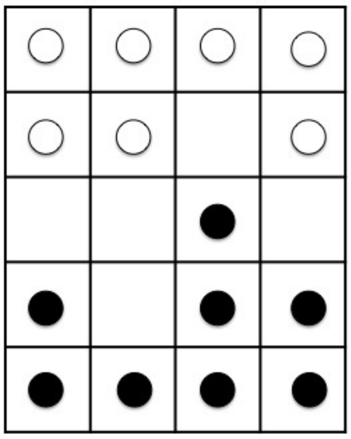
If it moves orthogonally, the board becomes like this:



The black pawn located on the fourth row and the second column can move on one of these three squares:



If it makes the catch, the board becomes:

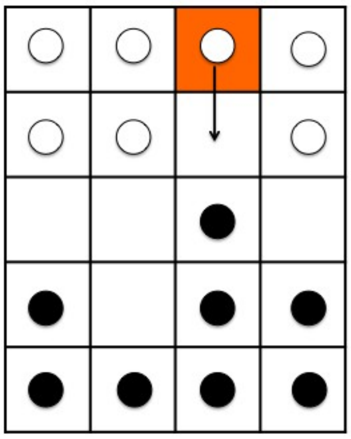


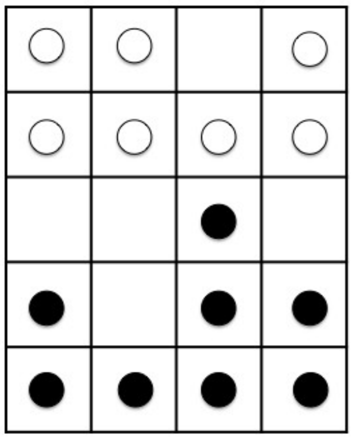
Etc.

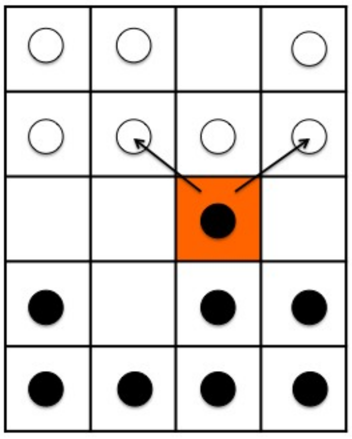
A player win the game as soon as he fills one of these two conditions:

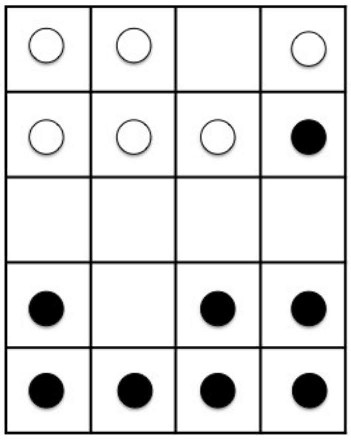
- One of his pawns moved to the opposite line to its starting position (bottom row for whites and top row for blacks).
- His opponent has no pawns.

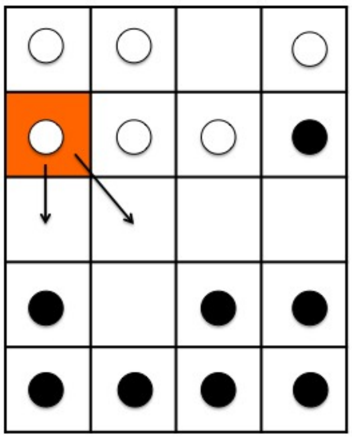
Example of victory with the rest of the previous game:

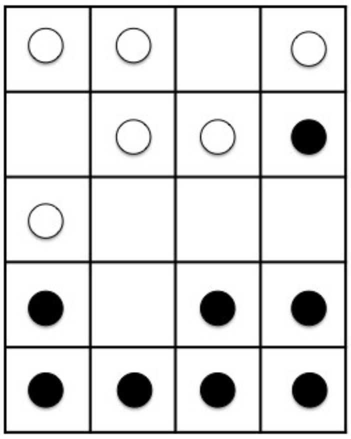


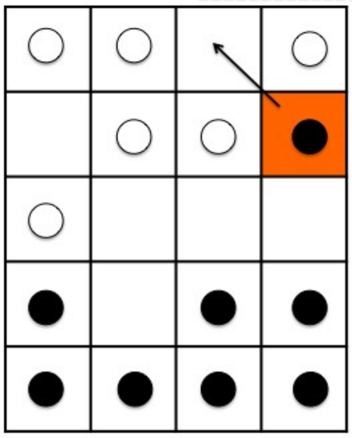


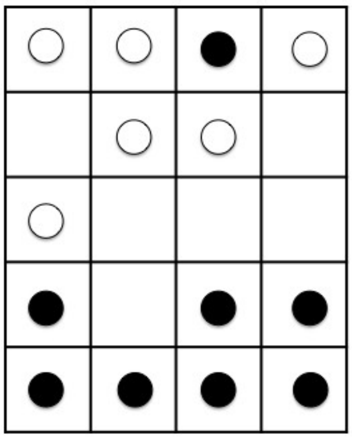












At the end we have a victory of blacks.

2.2 Programming this game in Python

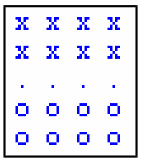
It is strongly recommended to read all the content of this part before coding.

Important notes :

- It may be possible to implement subroutines in addition to those asked.
- The board will naturally be a two-dimensional list of integers equal to 0, 1 or 2. An empty square will be represented by a 0, a pawn of the first player by a 1 and a pawn of the second player by a 2.
- Rather than recalculate regularly the dimensions of this list, we will prefer to put them into the parameters of our subroutines.
- The visual rendering of the program is only an indication, you can improve it.

Implement the following subroutines in a file "breaktrough.py ":

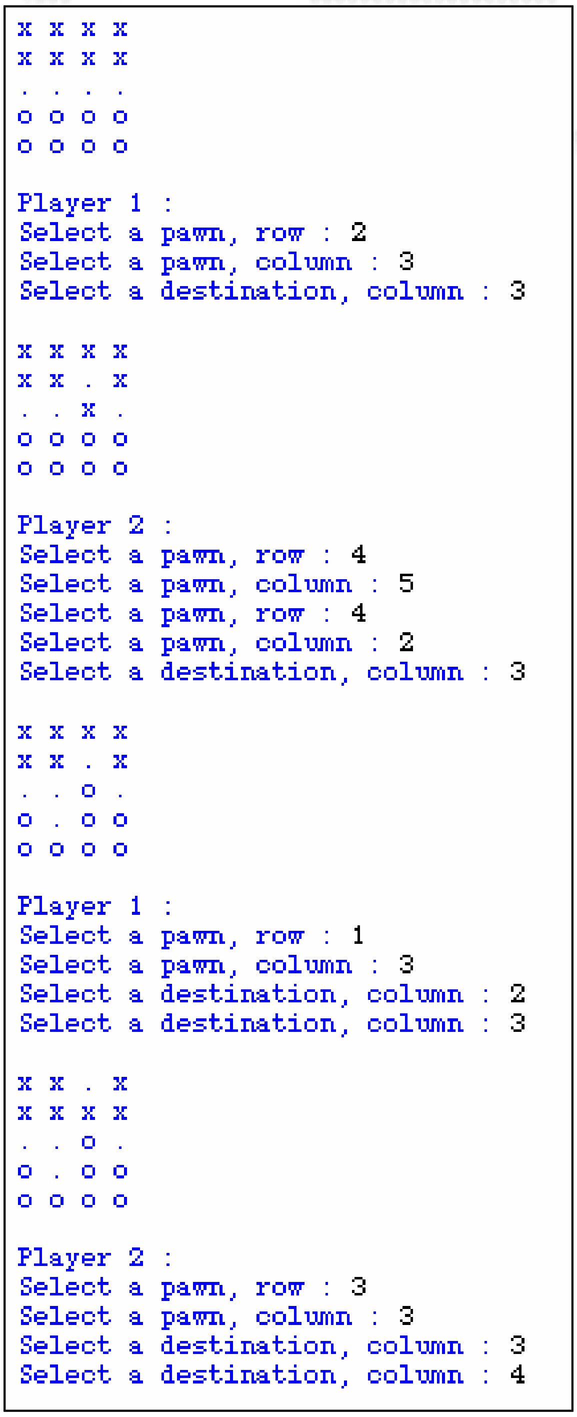
- A « newBoard(n,p) » function where **n** and **p** are positive integers, with **n>4**. It returns a two dimensions list that represent the initial state of a board composed of n rows and p columns.
- A « display(board,n,p) » procedure where **board is** a two dimensions list of integers equals to 0, 1 or 2, **n** represent his number of rows and **p** his number of columns. It displays the board on the console. We will represent an empty square by a '.', a white pawn by a 'x' and a black pawn by a 'o'. At the beginning, a board of 5 rows and 4 columns will be displayed like this:



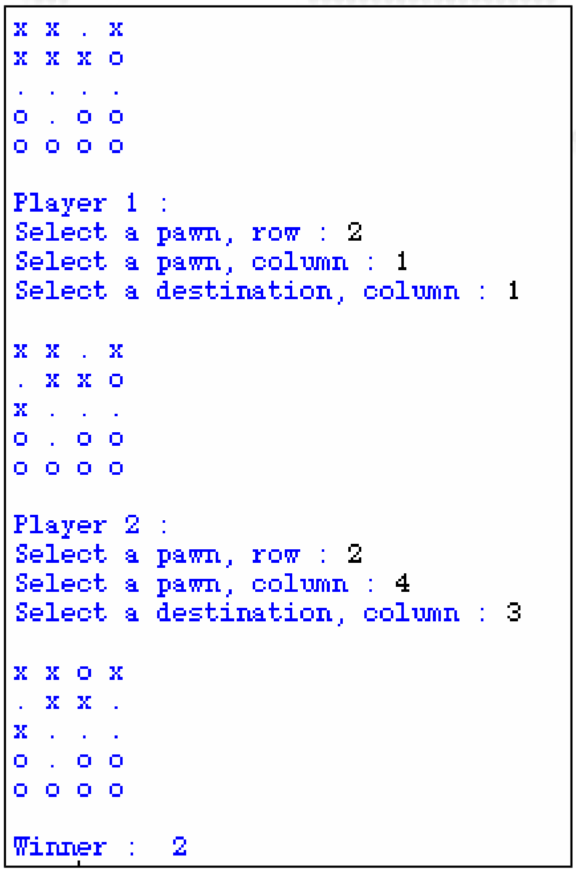
```
x x x x
x x x x
. . . .
o o o o
o o o o
```

- A « selectPawn(board,n,p,player) » function where **board** is a two dimensions list of integers equals to 0, 1 ou 2, **n** his number of rows, **p** his number of columns and **player** an integer that represent the player wich it's the turn (for example 0 for the first player and 1 for the second one). This function requires to the player to choose coordinates of pawn able to move. We will suppose that this pawn exists, and we will not test this fact here. While coordinates will not be valid under the rules of the game, we will ask him again to retry. Finally, the function will return those coordinates.
- A « where(board,n,p,player,i,j) » function where **board** is a two dimensions list of integers equals to 0, 1 ou 2, **n** his number of rows, **p** his number of columns and **player** an integer that represent the player wich it's the turn and **i,j** coordinates of a player's pawn. We suppose this pawn is able to move. This function requests to the player to write the column where he wants to put the pawn that we have just mentioned. While the number of the column will not be valid under the rules of the game, we will ask him to rewrite again. Finally, the function will return this number.
- A « breaktrough(n,p) » procedure where **n** and **p** are positive integers. This procedure will use the previous subroutines (and others if it is needed) to permit two players to play a full game on a board with n rows and p columns.

Again, here is the example of the subpart 2.1, but this time we use our program:



```
x x x x
x x x x
. . . .
o o o o
o o o o

Player 1 :
Select a pawn, row : 2
Select a pawn, column : 3
Select a destination, column : 3

x x x x
x x . x
. . x .
o o o o
o o o o

Player 2 :
Select a pawn, row : 4
Select a pawn, column : 5
Select a pawn, row : 4
Select a pawn, column : 2
Select a destination, column : 3

x x x x
x x . x
. . o .
o . o o
o o o o

Player 1 :
Select a pawn, row : 1
Select a pawn, column : 3
Select a destination, column : 2
Select a destination, column : 3

x x . x
x x x x
. . o .
o . o o
o o o o

Player 2 :
Select a pawn, row : 3
Select a pawn, column : 3
Select a destination, column : 3
Select a destination, column : 4
```



```
x x . x
x x x o
. . . .
o . o o
o o o o

Player 1 :
Select a pawn, row : 2
Select a pawn, column : 1
Select a destination, column : 1

x x . x
. x x o
x . . .
o . o o
o o o o

Player 2 :
Select a pawn, row : 2
Select a pawn, column : 4
Select a destination, column : 3

x x o x
. x x .
x . . .
o . o o
o o o o

Winner :  2
```

2.3A VERSION WITH A NOT REALLY CLEVER ARTIFICIAL INTELLIGENCE

Implement another version of this game (obviously, we will conserve the previous functional part), where a human will be able to play against the computer. The computer will play randomly.

3 FIRST ATTACK

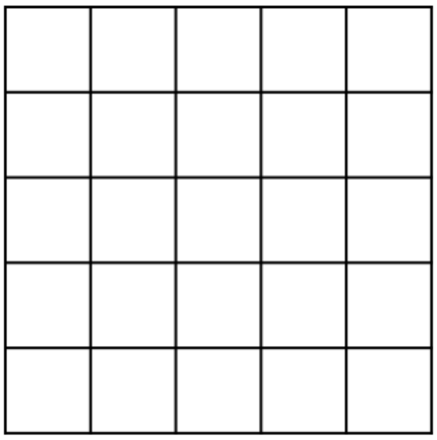
3.1RULES OF THIS GAME

Important note: no code is requested in this sub part, where we will explain the rules.

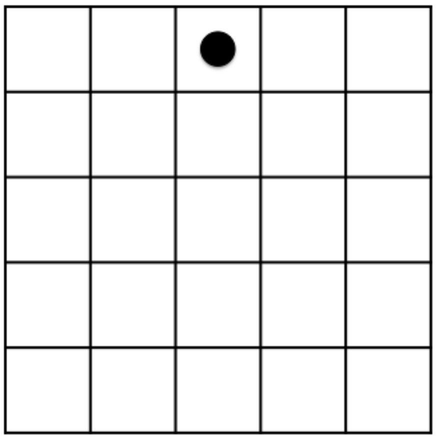
This game has been created by Frank Harary (Unknown date). Two players fight each other on a square board of **n** rows and **n** columns, initially empty. In the original rule, players play with the same pawns (we will propose a version where it is not the case).

In turn, players place a pawn on an empty square on the board so that the square will not be on the same row, the same column or the same diagonal of an existing pawn.
The winner is the last player able to place a pawn. In others words, when a player cannot place a pawn, he loose.

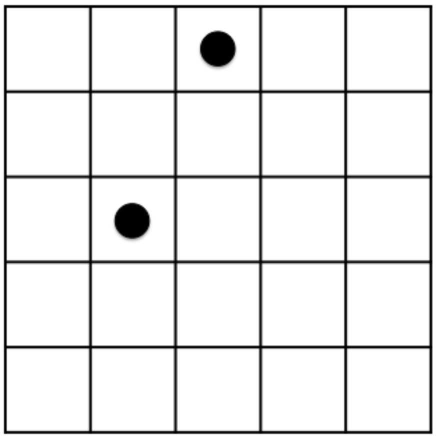
Example of game on a board of 5 rows and 5 columns:



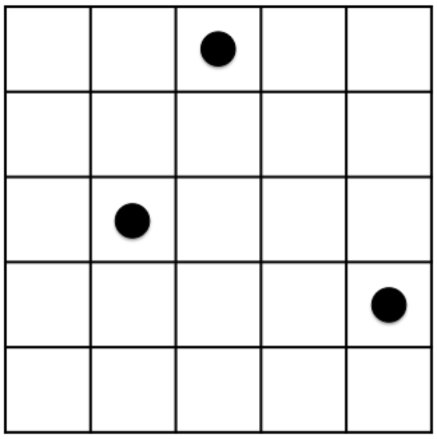
The first player's move :



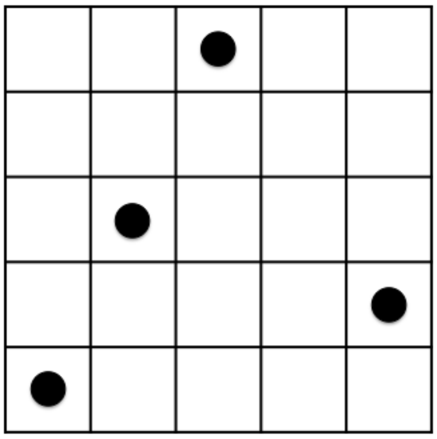
The second player's move :



The first player’s move :



The second player’s move :



The first player cannot pose pawns anymore, he therefore lost.

3.2 Programming this game in Python

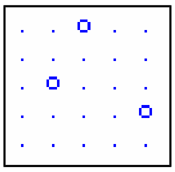
It is strongly recommended to read all the content of this part before coding.

Important notes :

- It may be possible to implement subroutines in addition to those requested.
- The board will be a two-dimensional list of integers equals to 0, 1 or 2. An empty square will be represented by a 0, a pawn by a 1 and a square in alignment with a pawn by a 2.
- Rather than to re-calculate the dimensions of this list regularly we prefer to pass the list into parameters of our subroutines.
- The example of visual rendering of this program is only indicative, you can improve it.

Implements following subroutines in a file “firstAttack.py“ :

- A « newBoard(n) » function where n is a positive integer. It returns a two-dimensional list that represents the initial state of a board of **n** rows and **n** columns.
- A « display(board,n) » procedure where **board** is a two-dimensional list of integers equals to 0, 1 or 2, and **n** his number of rows and columns. It displays the board on the console. We will represent an empty square by a ‘.’ And a pawn by a ‘o’. After many turns, we will get a display of this genre:



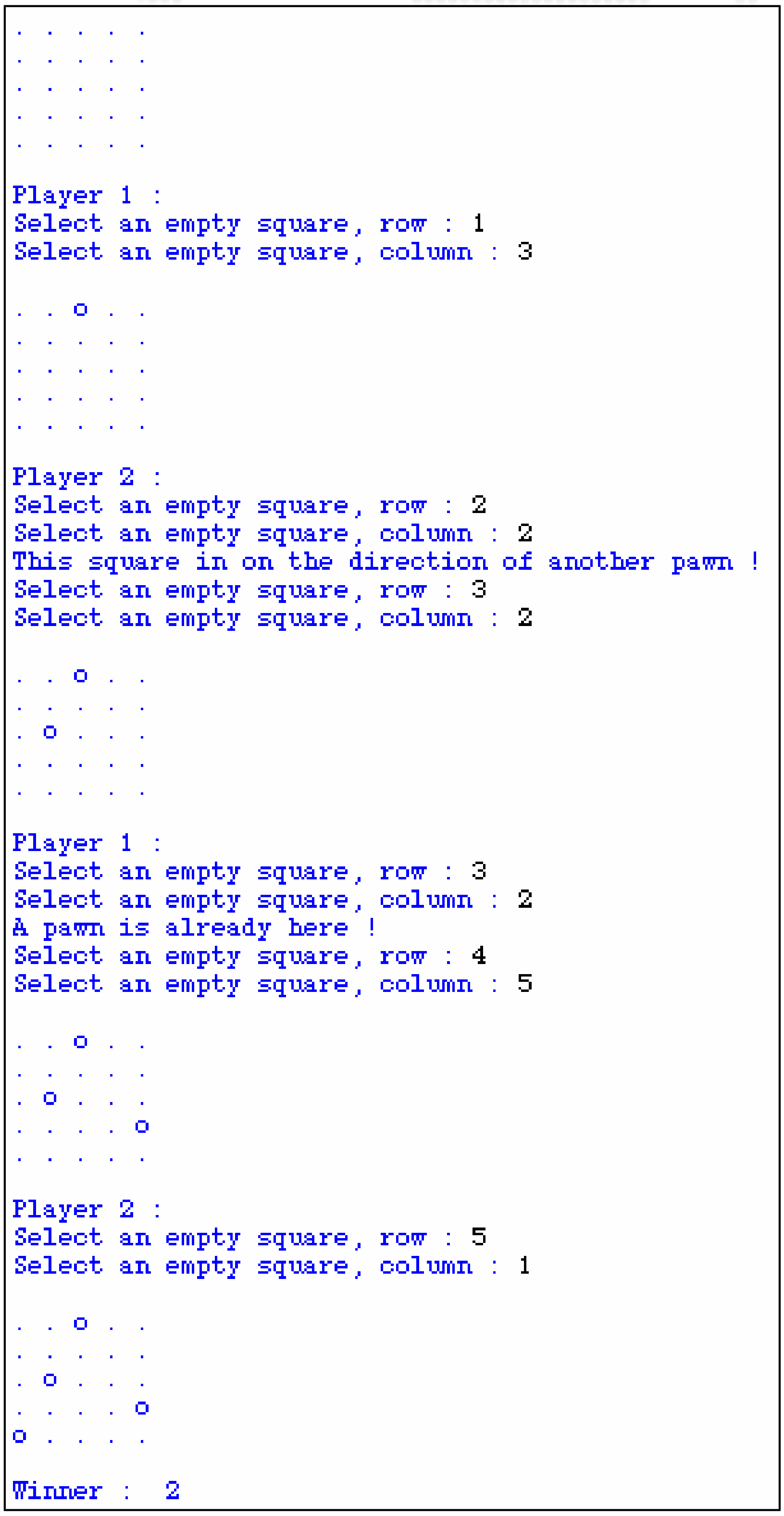
```
. . o . .
. . . . .
. o . . .
. . . . o
. . . . .
```

- A « notFinish(board,n) » function where **board is a two-dimensional list of integers equals to** 0, 1 ou 2, and **n** his number of rows and columns. It returns **True** if we are still able to pose a pawn on the board.
- A « selectSquare(board,n) » function where **board** is two-dimensional list of integers equals to 0, 1 or 2, and **n** his number of rows and columns. This function requests to write the coordinates of an empty square able to get a pawn. We will suppose that this square exists, and we will not test it here. While these coordinates will not be valid under the rules of the game, we will ask him again to retry. In case of bad entry, we will indicate the reasons (see example below). Finalement, la fonction retournera ces coordonnées.
- A « update(board,n,i,j) » procedure where **board** is a two-dimensional list of integers equals to 0, 1 or 2, **n** his number of rows and columns,

and **i,j** the coordinates of an empty square able to welcome a pawn. This procedure updates the board after the pose of a board on this square.

- A « firstAttack(n) » procedure where **n** is a positive integer. It will use the previous subroutines (and others if needed) to allow to players to play a full game on a board of **n** rows and **n** columns.

Here is the example of the sub part 3.1 again, but this time with our program:



```
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .

Player 1 :
Select an empty square, row : 1
Select an empty square, column : 3

. . o . .
. . . . .
. . . . .
. . . . .
. . . . .

Player 2 :
Select an empty square, row : 2
Select an empty square, column : 2
This square in on the direction of another pawn !
Select an empty square, row : 3
Select an empty square, column : 2

. . o . .
. . . . .
. o . . .
. . . . .
. . . . .

Player 1 :
Select an empty square, row : 3
Select an empty square, column : 2
A pawn is already here !
Select an empty square, row : 4
Select an empty square, column : 5

. . o . .
. . . . .
. o . . .
. . . . o
. . . . .

Player 2 :
Select an empty square, row : 5
Select an empty square, column : 1

. . o . .
. . . . .
. o . . .
. . . . o
o . . . .

Winner :  2
```

3.3 Another version

Implements another version of this game (obviously, we will keep the previous one), where each player has pawns of different colours, and where the constraint of alignment concerns only on pawns with the same colour.

4 PLEIADIS

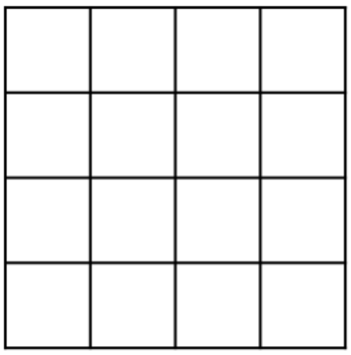
4.1Rules of this game

Important note: no code is requested in this sub part, where we will explain the rules.

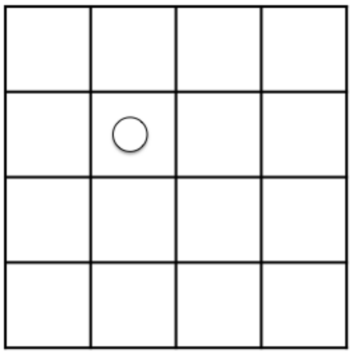
This game has been created in 2002 by Christian Watkins and Jan Kristian Haugland. Two players fight each other on a square board of **n** rows and **n** columns. The first player gets white pawns and the second one gets black pawns. At the beginning of the game, the board is empty.

Whites begin, then, players pose in turn their pawns on an empty square of the board in such way that for this square, the number of adjacent pawns of the opponent (orthogonally or diagonally) on the board is less than or equal to the number of adjacent pawns of the player.
The winner is the last player able to pose a pawn. In others words, as soon as a player is not able to pose a pawn, he loses.

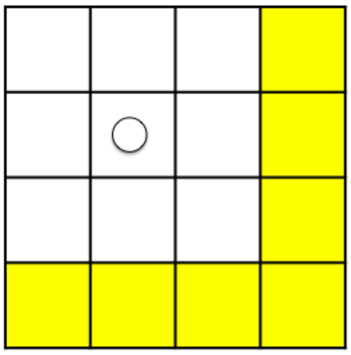
Example of a game on a board of 5 rows and 5 columns:



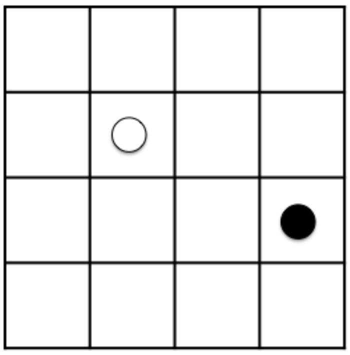
Placing of the first white pawn, no constraints because the board is empty:



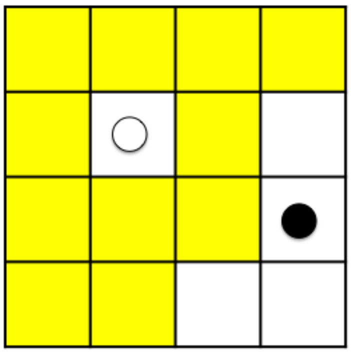
Here are the squares where the second player is able to pose a pawn:



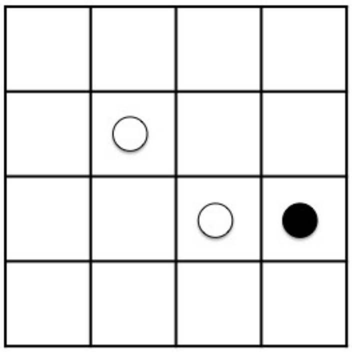
Placing of the first black pawn:



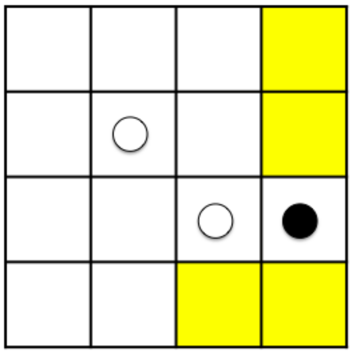
Here are the squares where the first player can pose a pawn:



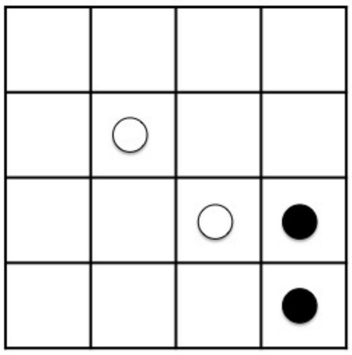
Placing of the second white pawn:



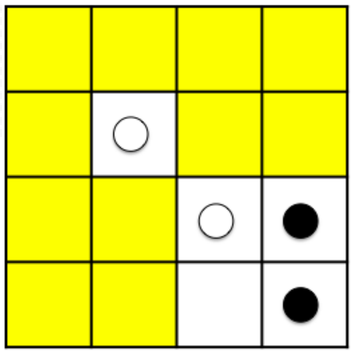
Here are the squares where the second player can pose a pawn:



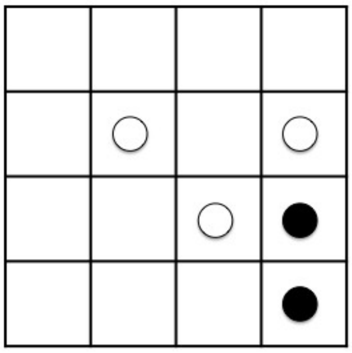
Placing of the second black pawn:



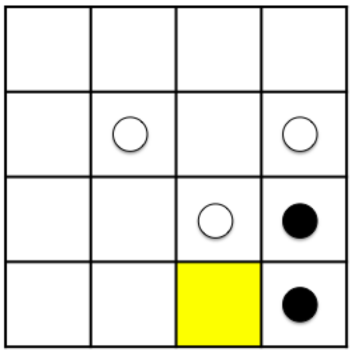
Here are the squares where the first player can pose a pawn:



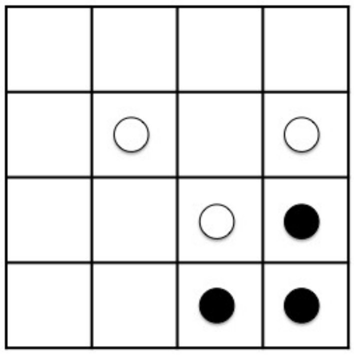
Placing of the third withe pawn:



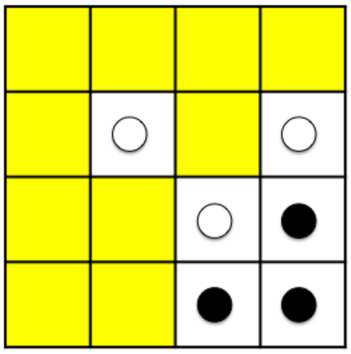
Here is the square where the second player is able to pose a pawn:



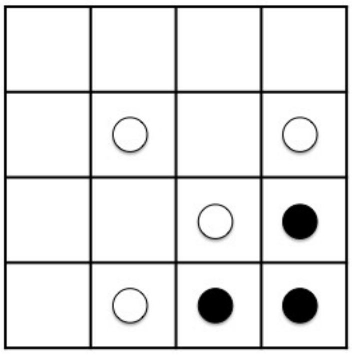
Placing of the third black pawn:



Here are the squares where the first player can pose a paw:



Placing of the fourth white pawn:



The second player cannot pose pawn, therefore he loses.

4.2 Programming this game in Python

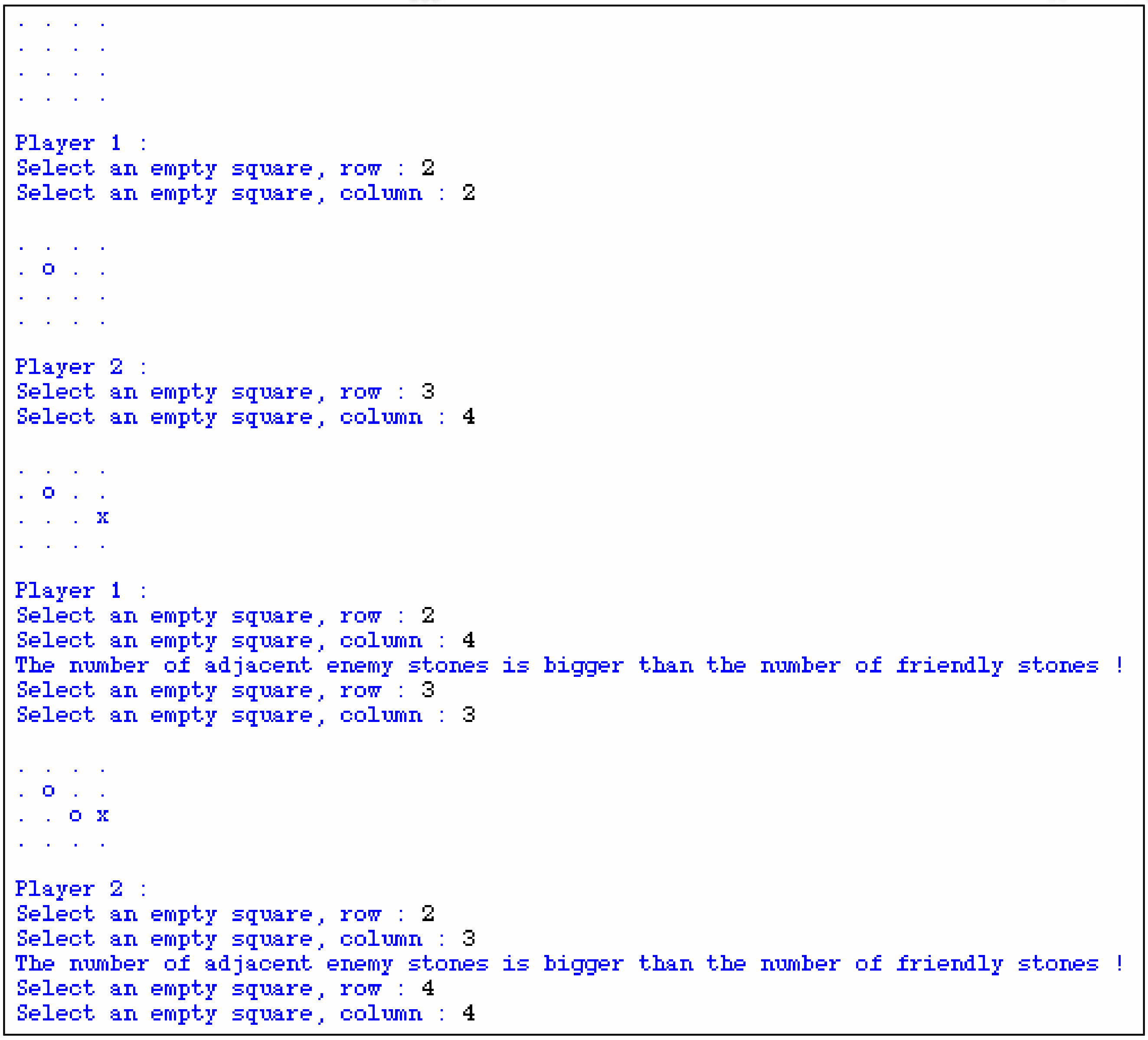
It is strongly recommended to read all the content of this part before coding.

Important notes :

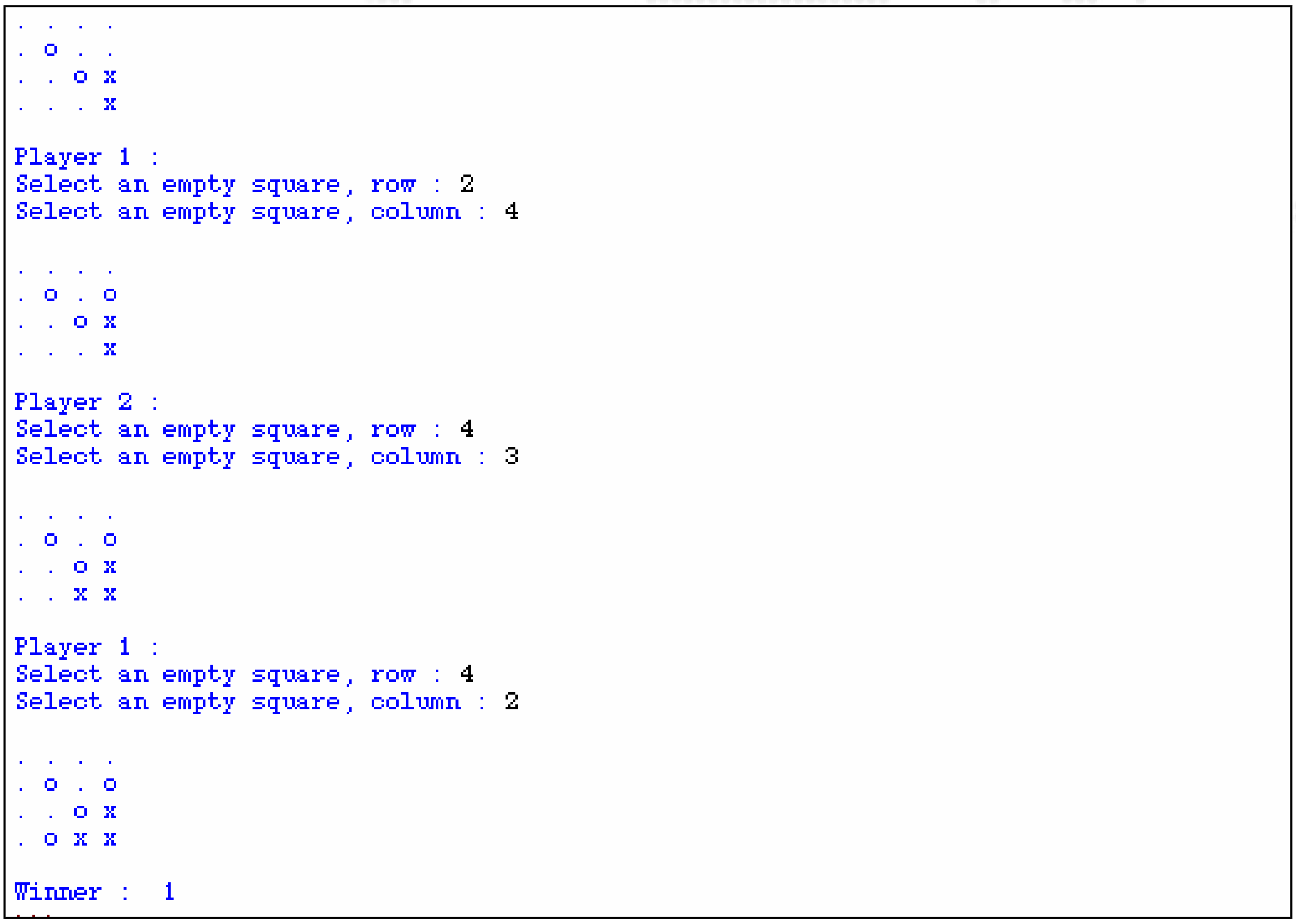
- We will eventually implement subroutines in addition to those requested.
- The board will be obviously a two-dimensional list of positive integers equals to 0, 1 or 2. An empty square will be represented by a 0, a pawn of the first player by a 1 and a pawn of the second player by a 2.
- Rather than to re-calculate the dimensions of this list regularly we prefer to pass the list into parameters of our subroutines.
- The example of visual rendering of this program is only indicative, you can improve it.

Implement in a file “pleiadis.py“subroutines allowing to play to this game. We will adopt a similar architecture of the code in the 3.2 sub part.

Here is again the 4.1 sub part, but this time with our program:



```
. . . .
. . . .
. . . .
. . . .

Player 1 :
Select an empty square, row : 2
Select an empty square, column : 2

. . . .
. o . .
. . . .
. . . .

Player 2 :
Select an empty square, row : 3
Select an empty square, column : 4

. . . .
. o . .
. . . x
. . . .

Player 1 :
Select an empty square, row : 2
Select an empty square, column : 4
The number of adjacent enemy stones is bigger than the number of friendly stones !
Select an empty square, row : 3
Select an empty square, column : 3

. . . .
. o . .
. . o x
. . . .

Player 2 :
Select an empty square, row : 2
Select an empty square, column : 3
The number of adjacent enemy stones is bigger than the number of friendly stones !
Select an empty square, row : 4
Select an empty square, column : 4
```



```
. . . .
. o . .
. . o x
. . . x

Player 1 :
Select an empty square, row : 2
Select an empty square, column : 4

. . . .
. o . o
. . o x
. . . x

Player 2 :
Select an empty square, row : 4
Select an empty square, column : 3

. . . .
. o . o
. . o x
. . x x

Player 1 :
Select an empty square, row : 4
Select an empty square, column : 2

. . . .
. o . o
. . o x
. o x x

Winner :  1
```

4.3A VERSION WITH AN ARTIFICIAL INTELLIGENCE PRETTY CLEVER

In this sub part, we will restrict us with square boards where their number of rows and columns are odd numbers. Propose a strategy where the first player playing is certain to win.

Indication : The first player will pose his pawn on the square of the center of the board (it is possible because his dimensions are odd numbers) then we will “trace“ the hits of his opponent. It is your goal to give meaning to the word "trace" used here.

Implement another version of the game Pleiadis (obviously, we will keep the previous one), where a human will be able to play against the computer that use the artificial intelligence described in this sub part.

5 GRADING SCALE

This scale can change, it is only **indicative**.

- Part 2: 12 points
- Part 3 : 12 points
- Part 4: 16 points

This makes a total of 40 points brought on 20 points proportionally.