CentraleSupélec

PROGRAMMING PENTAGO WITH MATLAB
Programmation du jeu Pentago sur MATLAB

Gustavo **CIOTTO PINTON**
Marcelo **MARQUES FREIRE DE CARVALHO**

Laurent **BOURGOIS**

CentraleSupélec Promo 2016

Contents

1 Introduction

1.1 The Objective

This work focus on the implementation of an artificial intelligence able to play effectively the board game Pentago. The programming language that will be deployed to achieve so is Matlab. Most Pentago AIs are developed in C++, hence using Matlab would bring some innovation for there is little documentation on programming a Pentago AI with this language. Moreover, this choice has a pedagogical value: we have previously worked with Matlab and our advisor has done a Pentago AI of his own before, in such way that it would be easier to compare results and approaches for the evaluation of our work.

For simplicity reasons we will employ « he » instead of « he or she » on the writing.

1.2 What is Pentago?

« *Pentago* is a two-player abstract strategy game invented by Tomas Flodén. The Swedish company Mindtwister has the rights of developing and commercializing the product.

The game is played on a 6x6 board divided into four 3x3 sub-boards (or quadrants). Taking turns, the two players place a marble of their color (either black or white) onto an unoccupied space on the board, and then rotate one of the sub-boards by 90 degrees either clockwise or anti-clockwise. A player wins by getting five of their marbles in a vertical, horizontal or diagonal row (either before or after the sub-board rotation in their move). If all 36 spaces on the board are occupied without a row of five being formed then the game is a draw.» (Pentago, 2015).

1.3 Why Pentago?

When Pentago was introduced in 2005, it immediately attracted the interest of Computer Science community, as it presents itself as «[...] a two player, deterministic, perfect knowledge, zero sum game: there is no random or hidden state, and the goal of the two players is to make the other player lose (or at least tie).» (Pentago is a first player win, 2015). Games that hold those characteristics, as Chess and Go, are of great interest for the computer science and frequently of great popularity worldwide.

« But where does the motivation to examine games comes from? Herik et al. [43] described, the interest of Artificial Intelligence researchers in strong game-playing programs as an important goal for more than half a century now. "The principal aim is to witness the "intelligence" of computers. A second aim has been to establish the game-theoretic value of a game, i.e., the outcome when all participants play optimally." Heule et al. [19] added the motivational question: "Can artificial intelligence outperform the human masters in the game? ". Yet it is important to mention, that the methods to show intelligence and outperform human master can differ substantially from those for solving. But altogether, games seem to be an interesting site to show Artificial Intelligence. Moreover solving games applies the Artificial Intelligence methods to larger problems which may become computational puzzles. Even so the topic of solving may be seen as a toy application, it is perfect to show that large knowledge based problems are solvable.» (On solving Pentago, 2015).

By solving we mean the computational sense of the word: finding an optimal strategy that would either proof that a player (the first or the second to play) always win or that both may force a draw. The strength of a solution may be categorised in as stated in (On solving Pentago, 2015):

1. Ultra-weak: It is known what is the result of the game when it is perfectly played, although the strategy that leads to this result is not known.
2. Weak: A strategy that leads from the initial position(s) to the perfect-play result for both players is known.
3. Strong: For all possible to have positions, a strategy that leads to the perfect-play result is known.

We may observe that «strongly solve» implies «weakly solved» that implies «ultra-weakly solved».

Unfortunately, many games present such complexibility to evaluate the possible outcomes that it takes too much time to trace a strategy, hence rendering the solution unreachable with the contemporane technologie. As it presents some symmetries and «not so many» states, Pentago is more easily solved than the previous examples of Chess and Go.

« The 6x6 version of Pentago has been strongly solved with the help of a Cray supercomputer at NERSC. With symmetries removed, there are 3,009,081,623,421,558 possible positions. If both sides play perfectly, the first player to move will always win the game.» (Pentago is a first player win, 2015).

All in all, we may conclude that the solution of Pentago belongs to a class of problems (solving games with an AI) that has been subject of everlasting interest of the scientific community and general public. It is an already solved problem, but it has been solved employing a supercomputer. Moreover, there is little documentation on programming Pentago with Matlab instead of C++.

We want to produce with this work an efficient AI, employing Matlab language and conventional computers to run it. Finally, we want it to be fast, as it should be suited for tests against other available AI and human players, in such fashion that we want to make it play in less than a couple of minutes.

1.4 Work Planning

We will deploy the following phases of our project:

1. First, we will have a player vs player enabled Pentago.
    - Creation of a graphical interface that enables players to play a piece on an empty space when it is his turn.
    - Modification of this interface to allow players to rotate the board after their play.
    - Introduction of a «game over» clause, that ends the game as a player wins or there is a tie.
2. Second, we will introduce an artificial intelligence able to play against an external player (i.e. player vs AI enabled).
    - Insertion of a random play AI able to play against a player, i.e. obeying the rules of the game.
    - Research about possible solving algorithms to implement.
3. Finally, we will work on the optimization of our AI, rendering its algorithm more efficient and robust (i.e. strong AI).
    - Implementation of efficient solving algorithms.
    - Further tuning for optimal results.

In order to advance from a module of work to the next one, we will employ unitary and integration tests for the validation of the progress. Likewise, we will employ tests and simulations to verify the performance of our choices in the tuning phase of our algorithm, for instance confronting our artificial intelligence with the one programmed by our advisor.

2 Aspects of Development

In this section, we will explain and discuss the solutions we used to implement a Pentago game over *MATLAB*. Firstly, we dedicate a section to detail the main aspects concerning the graphical interface and its components. In this small section, we describe the general structure of the game, including the approach we have chosen to organize the progress of the program. Then, the next sections are dedicated to the heart of our program, which is the algorithm used to simulate an artificial intelligence.

It is important to say that only the most important parts of the program are fully explained. In order to visualize all code, please refer to the files which will be indicated in the next sections.

2.1 Graphical Interface - GUI

All the code used in this item can be found in file **pentago_plateau.m**

The window which can be seen in Figure 1 shows the graphical interface of the Pentago game. It is composed by two buttons (*'Start Game'* and *'Reset Game'*), a textbox (initially with '*Press [Start Game] to start.*'), whose function is to inform the player about the current state of the game, and a 6x6-slot matrix, which represents the board.

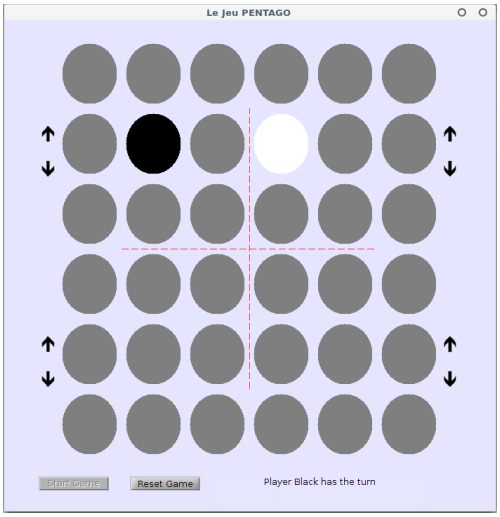


Figure 1: GUI of our game.

In *MATLAB*, a window is created by the command figure and its characteristics are defined by some parameters. In our case specifically, the variable plateau holds a reference for the window and the parameters define some attributes, such as height, width and screen positions.

```
% It creates a new figure which is going to host all the components
plateau = figure ('Visible', 'on', 'Position', [1500, 1000, width, height], ...
    'Name','Le Jeu PENTAGO ','NumberTitle','off', 'Resize', 'off', ...
    'Color', [0.9, 0.9, 1], 'MenuBar', 'none');
```

where the variables width and height are started with

```
% Board's characteristics
width   = 700;
height  = 700;
```

The parameters are detailed below.

1. *Visible* determines whether the window will be visible right next it is built.

2. *Position* is an array containing the coordinates x and y of the screen, width and height of the window, respectively.

3. *Name* is the string that will be displayed in the figure window.

4. *Number Title* set to 'off' means that the string 'Figure No' will not be displayed in the figure window.

5. *Resize* set to 'off' means that the window cannot be resized.

6. *Color* sets the background color of the figure. The array contains three values, which determine the color according to the RGB scale. In our case, for instance, 0.9 represents 90

Having defined the window, the next step is to determine the axis which the components are going to be placed in. For that, we use the command axes, as shown below.

```
% Board's characteristics
% axes1 represents the total space where we can place our
%       components into
axes1 = axes('Units','Pixels','Position',[50,75,(width - XMargin) , (height - YMargin) ],
    'XLim', [-30 (width - XMargin + 30) ], ...
    'YLim', [0 (height - YMargin) ], 'Color', 'none', 'Visible', 'off');
```

where *XMargin* and *YMargin* specify the empty space between the margins of the window and the margins of the axes. They are set to

```
XMargin = 100;
YMargin = 100;
```

In Figure 2, the role of this component is more clearly visible. In few words, an axes components plays the role of a 'painting area', which multiple components can be drawn on. The attributes *Color* and *Visible* specify the color of the axes back planes and the their visibility, respectively. Setting *Color* to *none* means that the axes is transparent and the figure color shows through. Setting *Visible* to *off* prevents axis lines, tick marks, and labels from being displayed. Attributes *XLim* and *YLim* specifies the minimum and maximum values of the respective axis.

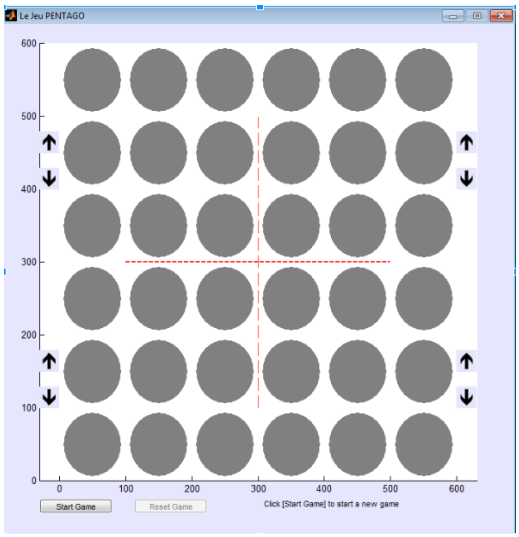


Figure 2: *Axis1* with *Color* and *Visible* different from *none* and *off*

The buttons *Start Game* and *Reset Game* are created by the command uicontrol, as shown below.

```
% Adds start and reset buttons
buttonStart = uicontrol ('Position', [50 30 100 20], 'String', 'Start Game', ...
    'Callback', @start_pressed);
buttonReset = uicontrol ('Position', [180 30 100 20], 'String', 'Reset Game', ...
    'Enable', 'off', 'Callback', @reset_pressed);
```

The most important parameter is *Callback*. It specifies a function handler, which will be activated when the button is pressed. In other words, the function *reset_pressed* is called right after a mouse click event over the respective button.

Despite the fact that each one of the 36 components that compose the board look like circles, they are obtained by the command rectangle, presented below as well.

```
for l = 1:6
    y = space / 2;

    for c = 1:6
        rectangles((6 -(c-1)),l) = rectangle('Curvature', [1.0, 1.0], ...
            'Position', [ x, y , rectWidth-space, rectHeight-space], ...
            'FaceColor', colorRectangle, 'EdgeColor', 'none',...
            'ButtonDownFcn', {@rectangle_pressed,  (6 -(c-1)), l});

        y = y + rectHeight;
    end

    x = x + rectWidth;
end
```

The circular shape is achieved thanks to the parameter 'Curvature', which specifies the curvature of both rectangle sides. This parameters is composed by a 2-element array, in which each one of the components can only vary from 0.0 to 1.0. They represent the fraction of width (or height) of the rectangle that is curved along the top and bottom edges. For example, 0.0 means no curvature at all and 1.0 creates an ellipse. If both are set to '1.0' and the two sides are equal, then we will have a circle because an ellipse having all axis with same length is a circle as well.

The parameter 'ButtonDownFcn' determines, as the parameter *Callback* in uicontrol, the handler of the function that will be called when a circle is clicked on. The only difference is that, in this case, we will send some parameters in addition to those that are predefined by MATLAB. l and c represent the indexes of each button in the matrix. In this way, when an event is detected, we are able to identify precisely which button has launched it.

At last, the eight arrows are created by the command image (only two represented below):

```
% The following lines are responsible for reading the images from the respective files.
% These images correspond to the arrows that make the boards to turn right or left.
rotate_up = imread('arrow_alt_up.jpeg');
rotate_down = imread('arrow_alt_down.jpeg');

% Adds all the eight arrows
textQ1L   = image(-30,450,rotate_down, 'ButtonDownFcn',{@turn_pressed, 1, 'R'});
textQ1R   = image(-30,400,rotate_up, 'ButtonDownFcn',{@turn_pressed, 1, 'L'});
[...]
```

The command imread reads an image from a file and *ButtonDownFcn* has the same behavior as the cases treated above. It should be noted equally that turn_pressed receives as parameters the quadrant (1 in our case) of the respective arrow and the direction ('R' or 'L') it represents. '-30' and '450' are the x and y coordinates of the first arrow respectively and '-30' and '400' represent the same for the second one.

2.2 Game's Progress

Once we have programmed all the required components for the GUI, we need to determine how the program will behave, that is, how each of them will affect the game's progress. For that, we need to define some important variables that are going to represent the game's state:

```
% resets matrix of states.
state_matrix = zeros(6,6);
turn = 'N';
hasTurned = 0;
hasPlayed = 0;
isComputer = 0;
empty_Slots = 36;
```

1. *turn* specifies which player has the current turn. Initially, the turn has no owner, then it is set to 'N', standing for Nobody. 'B' stands for the Black player (by convention, representing the computer) and 'W' for White (human player).
2. *state_matrix* represents the matrix which contains the state of every position of the board.
3. *empty_Slots* keeps the number of available slots.
4. *hasTurned* and *hasPlayed* indicate whether the current player has turned one of the sub boards and made a move respectively.
5. *isComputer* indicates if the computer has the current turn. Obs.: rigorously we do not need it, because we have already the variable 'turn' controlling the game. However, if, in a not distant future, we desired to allow the computer to control the white pieces, we would require this variable.

In order to begin a new match, the human player needs to push the button *Start Button*. This action will launch an event, that will be processed by the function *start_pressed*. At first, the button *Reset Game* is enabled and *Start Game* disables itself. In other words, it means that a game can only be reset after the it is started. Next, the function determines randomly, through the command 'randi', whether the human player or the computer should start playing. In the first case, we update the state variables to reflect that decision and wait that player to choose one of the free slots.

When the human player hits one of the circles, another event is created and the function rectangle_pressed assumes the control of the execution. Firstly, it checks if that position is really available and if it is not, an error message is written in the text space. If the player tries to turn the board before playing, an error message will be generated as well. In contradiction with all previous statements, the function sets the respective position of *state_matrix*, repaints the board of the figure (function *refresh_board*) and enables the player to turn the board as he wishes by setting the variable *hasPlayed* to 1.

The human player is allowed then to press one of the eight arrows along the board. Doing so, he produces an event, which is processed by the function *turn_pressed*. This routine verifies if the player can do this and, in the positive case, the matrix state and the board are updated (functions *rotate_quadrant* and *refresh_board* respectively) and *hasTurned* is set to 1. Finally, it calls another function, *verify_turn_changing*, which is responsible to verify some conditions, such as whether the game has finished or there is not available slots anymore, and to change the turn if nobody has won yet (function *change_turn*). In this last case, the function calls then one of the routines which implement an algorithm of artificial intelligence (see next sections), which returns the move, quadrant and sense of rotation that the computer has chosen to make. The last step is to update the state variables and repaint the board.

If the computer has been selected to start the game, then this last process is done before any move of the human player.

The human player can press the *Reset button* at any time. When this happens, the function *reset_pressed* is called. This routines restores all state variables and board to their initial conditions, enables the *Start button* and disables itself. In the way as before, a game can only be started if it has been reset.

2.3 Evaluating a board

All code used in this section can be found in file **evaluate_board.m**.

The best algorithm ever created would not be enough if we did not have a way of evaluating a board efficiently. For that reason, we developed a method capable of detecting the presence of lines, columns and diagonals contained in the matrix of states. This function is named *evaluate_board* and receives as parameters the state matrix and the player for whom the board will be evaluated.

This routine uses essentially six variables to count the total score of a board:

```
hasBeenDetected = zeros(6,6);
playerScores = zeros (4,1);
opponentScores = zeros (4,1);
scores = [playerScores  opponentScores];
foundUltraCondition = 0;
value = 0;
```

1. *value* is the result that will be given to that board.
2. Each element of *hasBeenDetected* is an integer whose representation in binary determines if that element has already been detected taking place in at least one line/column or diagonal. 0001 means that it takes place in a COLUMN, 0010 in a LINE, 0100 in a DIAGONAL to the right and 1000 in a DIAGONAL to the left. For example, if hasBeenDetected(i, j) is 0011, it means that position forms a LINE and a COLUMN at the same time.
3. *playerScores* is an array that stores the scores of all combination the player's got so far in each one of the possible configurations (COLUMN, LINE and DIAGONALS). For example, if the player has a 3-element line, then playerScores(2,1) receives the score for that combination.
4. *opponentScores* is the same but for the opponent.
5. *scores* is the matrix containing both playerScores and opponentScores.
6. The column concerning player is the first one and the column concerning the opponent is the last one. Its use is going to be discussed in the following lines.
7. *foundUltraCondition* is true if we detect a line, column or diagonal completely filled with 5 pieces.

Once that the variables were initialized, the routine verifies whether each filled position (x,y) of the matrix of states forms a possible combination. In order to do that, we introduce another index, *offset_index*, which starts at 4 and is decremented by 1 every iteration. This process stops when *offset_index* reaches 0. It is used to retrieve a line, column or diagonal of length equivalent to 1 + *offset_index*, as it is shown below.

```
t = state_matrix (x:(x+offset_index) , y);
```

It is evident that we need to verify that (x+offset_index) respects the dimension of state_matrix before executing the previous command.

Next, we need to verify if all elements of t are the same. For that, we execute the following command, provided by MATLAB:

```
if all(t == t(1)) [...]
```

If that condition is true, then we refer once more to *offset_index*. If it is equal to 4 and all elements are the same, then we do not need to verify the other elements, because this situation is equivalent to the maximum

or minimum score, depending on the value of *state_ matrix(x,y)* (if it is equal to *player*, it is maximum, minimum otherwise). We set then foundUltraCondition to 1, which is going to stop all loops (those ones that iterate x and y and that one which iterates *offset_ index*).

The number of filled positions and the score that the combination generates are then stored in the matrix scores, introduced before, the respectives columns depending on the player. The code below implements the solution.

```
scores(1, index_player) = scores(1, index_player) + 10^(3*(offset_index+1));
```

index_ player determines the columns that are accessed: if the current position we are verifying corresponds to the player, then *index_ player* is 1 and the first column is modified. If not, then *index_ player* is set to 2 and the second column of scores is accessed. It should be noted as well that all the scores are added up. So, the result of a board is as big as the number of combinations. In this way, the computer always prefers to increase the number of combinations (refer to section **Searching the best move**).

The last thing to do is to update hasBeenDetected with the positions of each element composing the combination. That prevents that the same element is counted two or more times for the same combination. For example, let's consider that the positions (i,j), (i, j+1) and (i, j+2) are filled and compose therefore a three-element line. (i, j) is iterated before the others and the score is given according to the size. If hasBeenDetected had not been update, then the program would counter (i, j+1) and (i, j+2) as another two-element line, what is evidently false because it has been already counted. For a line, each position of hasBeenDetected with same coordinates than the line present in *state_ matrix* is added up with 2 (0010 in binary).

```
% fills hasBeenDetected with 0010 in the
% respective LINE
hasBeenDetected (x , y:(y + offset_index)) =  ...
        hasBeenDetected (x , y:(y + offset_index)) + 2;
```

Additionally, if there is only one element that are not equal in the combination (for example, a column composed by 4 black pieces and only one white piece), then we attribute an intermediate score between the points that a complete combination would receive and the points that 4-element combination would have. This means that the computer would rather save its own skin before forming a 4-element configuration (refer to section **Searching the best move**).

The last thing the function does is summing up everything up to obtain the final result of the current board:

```
value =  value + sum(scores(:, 1) - scores(:, 2));
```

2.4 Searching for the best move

2.4.1 The choice of the algorithm

It may be trivial that, with infinite time available, one might easily solve most games just by exploring the full game-tree. As it is not the case in real world, one has to implement search algorithms that reduces the complexity of the problem and run on an acceptable amount of time.

There are many available methods for searching the best move in two players zero-sum games with perfect information, being the most commons: Alpha-Beta Search (and variants), Proof-Number Search (and variants), Threat-space Search and Retrograde search («On solving Pentago, 2015»).

For this project, we have chose to implement the Alpha-Beta Search, as it is the most common algorithm for solving Pentago. Moreover, this algorithm presents great many interesting variants, that are commonly employed on AIs, and therefore well comprehending him is a good asset to well understand the more elaborated solutions available.

As for the complexity of this approach, (Russell, Stuart J., 2003) explains that with an (average or constant) branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated (when the move ordering is pessimal) is $\mathcal{O}(b * b * ... * b) = \mathcal{O}(bd)$ – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about $\mathcal{O}(b*1*b*1*...*b)$ for odd depth and $\mathcal{O}(b*1*b*1*...*1)$ for even depth, or $\mathcal{O}(\frac{bd}{2}) = \mathcal{O}(\sqrt{bd})$. In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation. Meanwhile, (McCarthy, 2006) poses that when nodes are ordered at random, the average number of nodes evaluated is roughly $\mathcal{O}(b(\frac{3*d}{4}))$.

The Alpha-Beta Search or the Alpha-Beta pruning is based on the minimax algorithm but presents a new feature: it is capable of «pruning», skipping an area - subtree - of the search that is not capable of affecting the result of the search. The (Standford University, 2015) presents the following general description:

1. Consider a node n somewhere in the tree, such that one can move to that node.
2. If there is a better choice m either at the parent of the node n or at any choice point further up, n will never be reached.
3. Once we have enough information about n to reach this conclusion, we can prune it.

Alpha-Beta pruning gets its name from the following parameters:

α = the value of the best choice we have found so far at any choice point along the path for MAX.
β = the value of the best choice we have found so far at any choice point along the path for MIN

[...] Alpha-Beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current values of α and β.

It is possible to synthesize this behavior with pseudo-code - translation to English from the one available on (Élagage alpha-beta, 2015) in French - :

```
function ALPHABETA(P, alpha, beta) /* alpha always smaller than beta*/
    if P is leaf then
        return the value of P
    ifelse
        if P is Min node then
            Val = infinite
            for each children Pi of P do
                Val = Min(Val, ALPHABETA(Pi, alpha, beta))
                if alpha >= Val then  /* alpha cutoff*/
                    return Val
                endif
                beta = Min(beta, Val)
            endfor
        ifelse
            Val = -infinite
            for each children Pi of P do
                Val = Max(Val, ALPHABETA(Pi, alpha, beta))
                if Val >= beta then /*beta cutoff */
                    return Val
                endif
                alpha = Max(alpha, Val)
            endfor
        endif
    return Val
```

```
    endif
end
```

An execution of the alpha-beta pruning is illustrated below (Chalmers - GÖTEBORGS UNIVERSITET, 2015):

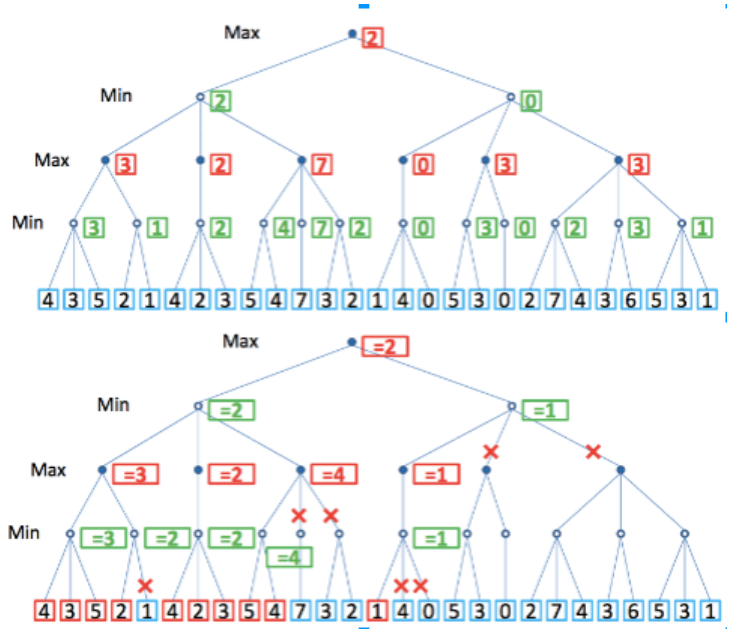


Figure 3: Illustration of an alpha-beta pruning run.

2.4.2 Coding

All code used in this section can be found in the file **alpha_beta_search.m**.

First, we establish what the function will return: the evaluation of the board(*eval_value*), the place([line column]) where the next piece will be played(move), the quadrant where the rotation is applied after the piece be placed (quadrant) and the direction of the rotation (direction). This may be encapsulated in an array using Matlab's succinct syntax.

```
function [eval_value move quadrant direction] = alpha_beta_search (state_matrix,
                                emptySlots, depth, alpha, beta, isMaximizing, player)
```

Second, we establish the base case of our recursive call, triggered by reaching the final depth of exploration. This is indicated by the variable depth when it reaches 0.

```
if depth == 0 || emptySlots == 0

    [eval_value] = evaluate_board(state_matrix, player);
    move = [];
    quadrant = 0;
    direction = 0;
[...]
```

Finally, we do the actions required for the alpha beta pruning when it does not find itself in the base case. If that is the case, we will firstly initialise the variables used in the alpha beta search (alpha, beta) and our

supplemental variable that allows us to do the pruning (mustStop). The supplemental variable will do so by breaking the while loops that are used to explore our tree, since $mustStop = 0$ will imply skipping further branches.

```
[...]
else
    dir = ['L' 'R'];
    mustStop = 0;

    if (isMaximizing == 1)
        eval_ = -inf;
    else
        eval_ = +inf;
    end
```

Then, we will proceed to explore our «tree» . In fact, instead of using a tree structure itself to explore the possible outcomes of the plays and implement the prunings, we will rather use a cascade of while loops that will break if we arrive to a natural limit of our tree - for instance, we will not explore the plays that may involve playing a piece in the 7th line of our board, since our board does not have a 7th line - or if the further branches are pruned by the alphabeta algorithm - that is signaled by the mustStop supplemental variable described above -.

The process begins by setting a piece of the color of the player (variable «player») which the turn is ongoing while the *alpha_beta_search* function is being used if we are in a maximizing phase or by placing a piece of the opposite colour if we are in a minimizing phase. After that, we will rotate one of the quadrants to one of the possible directions.

One might notice that the combination of line, column, piece played, quadrant of rotation and rotation direction plays a role analogue to a node in the tree structure.

```
[...]
l = 1;
while (l <= 6) && (mustStop == 0)

    c = 1;
    while (c <= 6) && (mustStop == 0)

        if state_matrix (l,c) == 0

            quad_index = 1;
            while (quad_index <=4) && (mustStop == 0)

                dir_index = 1;
                while (dir_index <= 2) && (mustStop == 0)

                    if isMaximizing == 1
                        state_matrix (l,c) = player;
                    else

                        if player == 'B'
                            state_matrix (l,c) = 'W';
                        else
                            state_matrix (l,c) = 'B';
                        end

                    end

                    state_matrix = rotate_quadrant (state_matrix, quad_index,
                         dir (dir_index) );
                    [...]
```

Afterwards, we will apply the *verify_ victory* function over the board that we are exploring to determine the state of the game, which may assume the values: white player is victorious('W'), black player is victorious('B'), we have a draw ('D') or the game is not yet over (0). The *verify_ victory* function may be found in the file **verify _victory.m**.

Knowing the state of the game, we may shortcut some decisions: if a player finds that by doing a certain move he will imply that the other player right after counter attack with a winning play, he will avoid the play that would led to his defeat by any means. He will not do it unless there are no possible play that will not lead to a defeat. That line of thinking is implemented by giving that board an evaluation so low that it can be outmatched by any non losing board.

In a similar way, if he finds a play that leads to his victory, he will no further explore the possible plays that may be done in that board: the play that he will do is already clear, we can not get better than a clear win. That is implemented by giving that board an evaluation so high it can not be outmatched by a non winning board.

Moreover, we will update the alpha and beta parameters following the rules previously explained that characterise the alpha-beta pruning. Furthermore, *mustStop* value will be update as well, in order to enable the pruning when it may be done.

```
[...]
state_game_ = verify_victory(state_matrix);

if isMaximizing == 1 && (state_game_ ~= 0) && (state_game_ ~= 'D')

    if state_game_ == player
        eval_MIN = +10^20;
        mustStop = 1;
    else
        [eval_MIN move_MIN quadrant_MIN direction_MIN] = ...
            alpha_beta_search(state_matrix, emptySlots - 1, depth - 1, alpha, beta, 0,
                player);
    end

    if eval_ < eval_MIN

        eval_ = eval_MIN;
        eval_value = eval_;
        move = [l c];
        quadrant = quad_index;
        direction = dir (dir_index);

    end

    if eval_ >= beta
        mustStop = 1;
    end

    alpha = max(alpha, eval_);


elseif isMaximizing == 0

    if (state_game_ ~= player) && (state_game_ ~= 0) && (state_game_ ~= 'D')
        eval_MAX = -10^20;
        mustStop = 1;
    else
        [eval_MAX move_MAX quadrant_MAX direction_MAX] = ...
            alpha_beta_search(state_matrix, emptySlots - 1, depth - 1, alpha, beta, 1,
                player);
    end

    if eval_ > eval_MAX
```

```
42
43          eval_ = eval_MAX;
44          eval_value = eval_;
45          move = [l c];
46          quadrant = quad_index;
47          direction = dir (dir_index);
48
49      end
50
51      if alpha >= eval_
52          mustStop = 1;
53      end
54
55      beta  = min(beta, eval_);
56  end
57  [...]
```

Lastly, we will restore the state of our board(*state_matrix*) to how it was before doing the play that we are analysing, hence enabling the board to be reused by the following play that we will analyse. This is done by doing the reverse of all modifications that we have applied since we run the alpha beta search on the board.

In addition to that, we will update our loops parameters and restart our loop.

```
1           if dir (dir_index) == 'L'
2                   state_matrix = rotate_quadrant(state_matrix, quad_index, 'R');
3           else
4                   state_matrix = rotate_quadrant(state_matrix, quad_index, 'L');
5           end
6
7           state_matrix (l,c) = 0;
8           dir_index = dir_index + 1;
9   end
10  quad_index = quad_index + 1;
11  end
12  [...]
```

2.4.3 Battle of AIs - A statistical point of view

In order to determine whether our algorithm is effectively good in terms of strategy and time, we wrote two other programs that basically simulate a match between two artificial intelligences - the one did and the other done by our advisor. The first program, **pentago_plateau_graphical_battle_of_IAs.m**, offers a graphical interface, which enables the user to follow every move taken by the algorithms. This function was adapted from the procedure explained in the previous sections, thus its characteristics are almost the same. The second one, **pentago_plateau_simple_battle_of_IAs.m**, does not provide a graphical interface but is able of simulating any number of matches in a row. Consequently, this function is faster and suits better a statistical study.

We decided to focus our analysis on two different parameters: the time which is required for each algorithm to decide its next move and the number of victories of each AI. In order to do that in the fairest way and considering that PENTAGO is a first-player game, each AI has the first move in exactly half of the 500 simulated matches, that is 250 matches. The code of this analysis is present in file **process_information.m**.

Concerning the number of victories, the player White, which represents our AI, won a total of 193 out of 500, while the player Black, our advisor's AI, succeeded in 223 matches. The rest of them, 84 matches, represents evidently the draws. The figure 4 summarizes these results. As we can see, the player White won 90 of its victories when the other player started playing and 103 otherwise. With respect to player Black, it won the most of its victories, 116, when it had the first move. The rest, 107 matches, were evidently won when player White had the preference. As for the draws, 44 were observed when player Black started and 40, when the other player had the first move.

Based on the previous results, we can observe that:

- Our advisor's algorithm presents a slightly better strategy than ours. Its chance of winning can be estimated at 44.6%, whereas ours can be estimated at 38.6%. Finally, a draw has the probability of 16.8% of occurring.
- The fact of starting the game has really an effect in the final result. Player White has a probability of 41.2% given that it started. The probability for player Black is slightly bigger: 46.4% of chance given that it's him that starts the match.

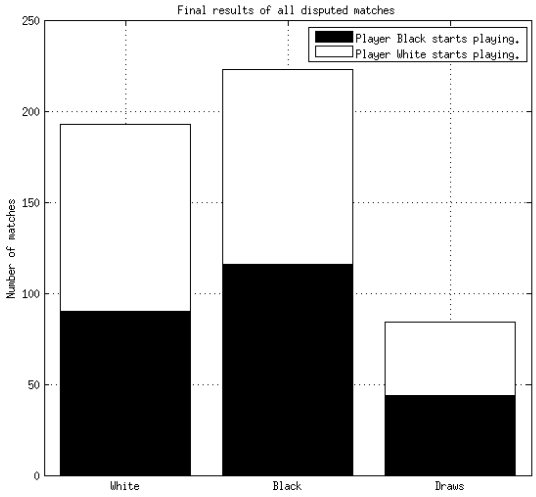
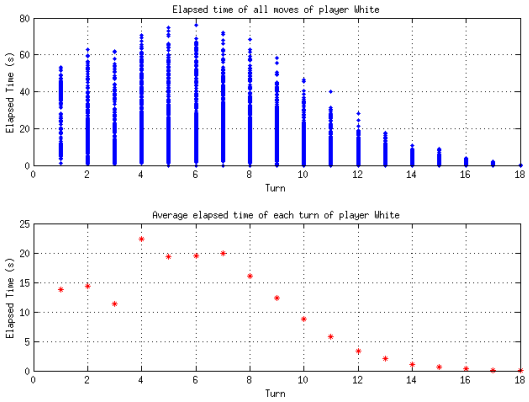
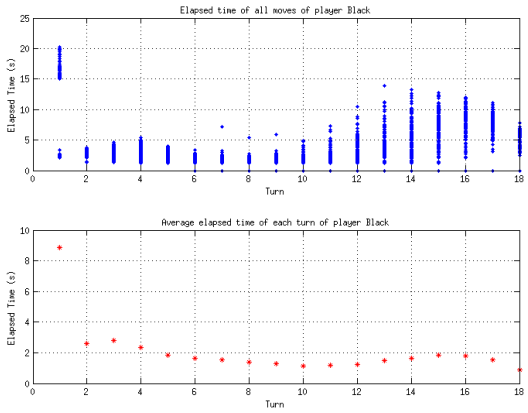


Figure 4: Number of victories of each player.

In relation to the time required to the calculations, the figures 5a and 5b show the results that were obtained for the 500 matches. We must consider that we used our own personal computers, that are not naturally designed to run simulations. Additionally, their resources were not dedicated exclusively for this experiment ('Midterm exams are coming!').



(a) Required time to Player White take a move.



(b) Required time to Player Black take a move.

The upper plots of each figure, in blue, are simple representations of the times which are saved right after each move was decided. In other words, each blue point stands for the time required to calculate every move. The plots containing red symbols show, for each turn, the average time required to decide that move. We observe then:

- Player White takes, in average, 9.5 seconds to decide. Player Black takes 2.05 seconds. The difference between these two times is explained by how which algorithm decides the move and evaluates a board. As explained before, our algorithm needs to verify each position of the matrix at least twice: once to evaluate if that position belongs to a row, column or diagonal, and another to conclude if that move is profitable.
- Player White's initial moves take longer times, meanwhile the delay of the latest moves decreases. That's already expected by the same reason we had before: at the beginning of the match, there are more positions to be verified, contrarily to the end, in which few available positions remain.
- Player Black's moves takes roughly constant time, around 2 seconds, to be decided. The only move that does not follow that property is the first one.

3 Conclusion

In order to program an artificial intelligence able to play Pentago, we have employed Matlab language, an alpha-beta pruning algorithm (first-depth) and a regular laptop computer. The results obtained reach our expectations:

1. Our graphical interface performed well. Easy to use, functional and enjoyable.
2. Our program may enable matches between humans, between AIs and mixed.
3. The processing time demanded by each play is reasonable.
4. The AI is efficient. It is roughly comparable in strength to the one developed by our advisor, which was set in the beginning of the project as our parameter of quality.

Nevertheless, an AI capable of strongly solving Pentago has been already programed with C++ and runned on a supercomputer at NERSC. Therefore, it is clear that may identify some room for improvement:

1. Programming Language: The use of C++ language instead of Matlab would enable better access to and better performance of the computational resources. For instance, it allows the algorithm used at NERSC to code and manipulate data in a very ingenious way, which enables a better solution.
2. Algorithm: From embodying transition tables to speed up the performance to increasing the depth of the alpha-beta search, there are some possible improvements. Some of them, for instance the increase of the depth of search, are not very well suited for an ordinary computer as it implies a greater overall processing time. The strong solving algorithm used at NERSC succeeded by associating the use of in game symmetries with brute force, all that being allowed by the way used to store data.
3. Hardware: NERSC's supercomputer, Edison, has « a peak performance of 2.57 petaflops/sec, 133,824 compute cores, 357 terabytes of memory, and 7.56 petabytes of disk » (Nersc, 2015). Meanwhile, our algorithm is developed to perform in laptops. It may be difficult to have access to a Cray XC30 supercomputer as Edison, but there cheaper and more accessible options on the market. For instance, one may exploit a regular computer GPU using CUDA to rise performances (NVIDEA,2015).

Lastly, those improvements might be hard to achieve for the time and the effort that they demand. They are maybe more suited for a longer and deeper project that surpasses a final graduation project. Furthermore, the results obtained on this project may be used by future students who want to solve Pentago or other board games on Matlab. They might also use our work, as well as our advisor AI, to compare different Pentago AIs. This last approach may allow some interesting results, for instance it may be used as a tool for evaluating non deterministic AIs.

4 Bibliography

- Pentago, (2015). Pentago. [online] Available at: http://en.wikipedia.org/wiki/Pentago [Accessed 14 Apr. 2015].
- Pentago is a first player win, (2015). [online] Available at: https://perfect-pentago.net/details.html#intro [Accessed 14 Apr. 2015].
- On solving Pentago, (2015). [online] Available at: http://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2011/Buescher_Niklas.pdf [Accessed 14 Apr. 2015].
- Standford University, (2015). [online] Available at: http://web.stanford.edu/~msirota/soco/alphabeta.html [Accessed 14 Apr. 2015].
- Élagage alpha-beta, (2015). [online] Available at: http://fr.wikipedia.org/wiki/%C3%89lagage_alpha-beta [Accessed 14 Apr. 2015].
- Chalmers - GÖTEBORGS UNIVERSITET, (2015). [online] Available at: http://www.cse.chalmers.se/edu/year/2014/course/TIN172/example-exam-solutions.html [Accessed 14 Apr. 2015].
- McCarthy, 2006. McCarthy, John (LaTeX2HTML 27 November 2006). "Human Level AI Is Harder Than It Seemed in 1955". http://www-formal.stanford.edu/jmc/slides/wrong/wrong-sli/wrong-sli.html [Acessed 20 Dec. 2006]
- Russell, Stuart J., 2003. Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2 http://aima.cs.berkeley.edu/
- Nersc, (2015). [online] Available at: https://www.nersc.gov/users/computational-systems/edison [Accessed 19 May 2015].
- NVIDIA, (2015). [online] Available at: https://developer.nvidia.com/how-to-cuda-c-cpp [Accessed 19 May 2015].