ARTIFICIAL INTELLIGENCE

COMPUTER CHESS PLAYER

Paolo Tagliani – ls26485

Gabriel Morera – is16528

Albert López – tl18112

Monday, December 19, 11

INTRODUCTION

Explained plainly, in this practice the only thing we have to do was to implement a computer chess player. In the implementation of the computer player was not implied the creation of the game interface so we only had to focus in the gameplay.

To achieve this objective the first thing we had to do was to know all the rules implied in the game of chess. These rules are briefly explained in the next section.

The next thing we had to think about was how we would be implementing the board representation and in which programming language we would implement it.

Once the board representation was done, we had to implement all the movement generation. This means that we had to implement all the possible movements that each piece can perform.

Afterwards, the search algorithm had to be implemented so we searched for optimizations of the classic alpha beta pruning.

Finally, after all this concepts had been implemented we had to create a well-defined heuristics that would endow our chess player with some kind of intelligence.

This order is the one used in this memoir to present all the concepts implemented in the practice.

BASIC CHESS SUMMARY

In this section the chess rules will be defined briefly. For a better understanding and a large explanation of these rules we reference the reader to the “Regles dels escacs” written by Albert Fornells Herrera and Elisabet Golobardes i Ribé.

The main objective of the game of chess is to capture the opponent’s king piece before the opponent captures yours. In order to achieve this objective you have sixteen pieces at your orders limited only by the different moves that each one can perform and by moves that the opponent perform against you.

Each player has eight pieces called pawns, two pieces called rooks, another two called knights, two more called bishops, a queen and a king.

At the beginning, the pieces are positioned as shown in the next picture.

Starting with the white pieces, the eight most advanced pieces are the pawns. The rooks are these pieces positioned in the corners of the board. Next to the rooks we have the knights, and aside the queen and the king we have the bishops. Finally, the king is the one with the cross and the queen is the remaining piece.

Besides the normal movements of each piece the chess game had many special movements as the castling and the pawn enpassant.

In addition of all this rules, the games are no always about winning or losing. The two players can end up even without any player winning or losing the game.

All these movements, special movements and special situations are very well explained in the document mentioned earlier. Explaining all this rules is not the focus of this document.

CHESS COMPLEXITY

Those who know how to play chess are conscious about the complexity of it. But the truth is that those who are not familiar with mathematic terms would never imagine the real complexity of the game.

Actually the computational power of our era is not able to handle all the possibilities involved in a normal chess game. Is for this reason that is not possible to perform what is called the perfect game. The perfect game is the game in which the player who plays with the white pieces is always able to win or even the game with a probability of one.

SPACE COMPLEXITY

Just to clarify how complex chess is we are going develop an intuitive approach to calculate the space complexity involved in a chess game. To follow this reasoning one must understand the basics of tree search.

Let's imagine that the branch number for each node is identified with *b.*

In the first level we will have b branches:

$$T(one\ expansion) = b$$

In the second level:

$$T(second\ expansion) = b + b \cdot b = (b+1) \cdot b \leq (b+1)^2$$

In the third level:

$$T(third\ expansion) = b + b \cdot b + b \cdot b \cdot b = \big((b+1) \cdot b + 1\big) \cdot b \leq (b+1)^3$$

And so on.

From these expressions one can infer that the space complexity of a chess game will never be greater neither equal to $(b+1)^m$.

Therefore, the chess space complexity is exponential.

TIME COMPLEXITY

As with the space complexity, the time complexity of a chess game is not affordable by our actual technology.

We will develop another basic approach to understand the real complexity of a chess game. Just to challenge the reader, we advance that is not possible to compute all the possible nodes of a chess game even if we had all the computers in the world working together while, what is called, the whole universe age[1].

Let's imagine that we dispose of 10^8 computers, what means that we dispose of 100 millions of computers working together.

We also suppose, which is a lot of to suppose, that every computer can generate 10^9 nodes per second.

[1] http://en.wikipedia.org/wiki/Age_of_the_universe

Finally we can approximate each game with an average of 40 moves per game and each move has an average of 30 moves starting from it. As we have already explained in the previous section, the chess space complexity is of b^m, so an average game would have 30^{40} nodes.

Multiplying the number of available computers by the number of total nodes processed by a computer in one second, we have that all the computers in the world would generate $10^9 \cdot 10^8 = 10^{17}$ nodes in a second.

So, how much time we will need to compute all the possible movements in a chess game?

$$TotalTime = {}^{30^{40}}/_{10^{17}} = 1.216 \cdot 10^{42} seconds =$$

$$3.855 \cdot 10^{34} years < 2.8 \cdot 10^{24} years = Universe\ age$$

Imagine now how complex chess is.

SOLUTIONS

Even if a complete chess game is not affordable for any computer in the world, the computer scientist community has found many ways to deal with such complexity.

The basic way to deal with complexity is to finish to expand nodes when a certain time has past. Is for this reason that all computer chess engines just search with a certain depth in the search tree.

Another approach is to prune those movements that you know beforehand that won't be better than the ones already expanded, given that your opponent will always choose the movement that is worst for your strategy.

Another strategy is to predefine the moves done given the actual moment in the game. So in the opening game – at the beginning- we can choose a sequence of moves established as the best moves for this game phase.

There are so many studies that try to reduce the process time needed to make a perfect chess player, but, even if this is not possible with our actual knowledge, at least has been achieved a computer chess player with a grandmaster level. Or what is the same, a computer chess player able to challenge the best human players.

BOARD REPRESENTATION

One of the most important aspects in this practice is the way that all current game data should be represented. Speaking about board representation not only is referred to how the board state will be implemented but all the necessary data to take the correct decision each moment.

So the first step we took was to decide the data that our board representation would contain and, afterwards, how this data would be represented in order to achieve a great efficiency improvement. Of course, before of how the data would be represented we chose the programming language that best fits our purposes.

BOARD REPRESENTATION CONTENTS

Even if we knew that, at first, we wouldn't have the knowledge enough to spot all the variables we were going to need, we realized that a great effort must been done in order to make a great design to not modify our base code while the practice was progressing.

Therefore we spent a lot of time thinking about all the information that would be needed to make a good decision for every done move.

Ignoring how all the next variables are implemented, we proceed to explain why are they needed and how they would help us to make the game go on.

First of all, we were completely sure that would be required some kind of structure to represent all the existing pieces in the game board. This data is represented by two variables called ***whitePieces*** and ***blackPieces***. Is of common sense that if you can obtain all the pieces involved in the game you could not continue with the game.

The next thing that we thought was if we were calculating the current state of the total pieces at each moment without storing its value in a well defined variable contained in the board representation or if we were to waste some space creating a few variables in order to clarify the code and made it more usable.

Despite the emphasis we have in improve the efficiency of our code, we thought that these variables would be used constantly, so, eventually, we decided to include them in the board representation. These variables were called ***totalWhitePieces***, ***totalBlackPieces*** and ***occupiedSquares*****.** As can be inferred, totalWhitePieces and, his analogous, totalBlackPieces are the total amount of existing white and black pieces in the board respectively. The variable occupiedSquare is not more than the total amount of existing pieces in the board, regardless his color.

Because we tried to enforce the non-use of global variables, the assigned color to our player should be contained in the board representation so as to be past to all the functions that need it. In our board representation this variable is called ***myColor*****.**

All these variables were those that came into our minds with facility and those that were implemented first. But there were many variables that were not intuitive enough to get rid of them in the first design phase. For this reason these variables were added to the board representation eventually, when we realize that we needed them.

One of these last variables was called ***boardState*** and represented all the possible moves that can be performed by the actual board state. Besides all possible movements given the actual board state, this variable also contains a variable called ***killedTypePiece*** that identifies if the given movement has killed an opponent piece and, if true, which is the type of the killed piece.

Afterwards, we realize that we needed to create a new variable to store the heuristic value of the movement represented by the actual instance of this board representation. So we created a variable called ***heuristic*** to store the value of the board state.

Finally, when we were working with the search algorithm we needed to know if we were generating a black or white move in order to generate the children correctly. For this reason we create the ***actualColor*** variable that identifies which children color must be generated in a given depth of the search tree.

We should note that in this section we only reflect all those variables that have a functional use and we don't speak about the types of these variables, or the data structures needed to achieve a correct use of them.

CHOICE OF THE PROGRAMMING LANGUAGE

Although this was not a difficult decision, our focus of discussion was between two very similar programming languages. C and C++.

But first of, how did we remove all the other options?

Even before starting to program we have in mind to optimize the performance of our algorithm. So for this purpose, besides a great algorithm design, we had to choose a really fast programming language. Therefore all scripting languages were removed from our choice options.

Furthermore, all those functional or logical oriented programming languages such as Lisp or Prolog were discarded because of our limited knowledge over them and because we have knew that the practice would be really complex.

After thinking a while and discarding all languages we really don't control, three options remained. C, C++ and Java.

The first option being discarded was Java. It was not an easy decision because with Java you have all that object facilities such as lists, hash tables, etcetera as a built in types. These objects would improve our code time and will assure that its implementation would be

correct, but despite all those facilities, we were focused in achieve a great performance result and a language running in a virtual machine would not be the best option to optimize it. So we discarded Java.

Eventually we ended up with C and C++. As before, we had the facilities given by C++ such as all the built in types it provides as Java. Of course, another advantage of C++ against C is his object oriented capacity. It's a known fact that using an object oriented programming language allows a better comprehension of the written code and, of course, with C++ you can achieve a similar performance to the C generated code.

So why did we not choose C++?

Because we are engineers. We are not afraid of not understanding the written code. Because, in this case, we prime the efficiency against the readability. And if we are not scared about the hardness of following the flow of our code, why in the earth we have to waste our processor time building all the required data to create a class object? As engineers we already know how a compiler works, and the internals needed to create, for example, a class. So why should we bother building a class when we know that all their data will be placed in the heap memory region, and, therefore, we will waste many precious milliseconds working with a double linked list[2] instead of taking advantage of this time by generating one more child in the search tree? Because readability? No. Not in our case.

So our last decision was to write C code.

BOARD REPRESENTATION IMPLEMENTATION

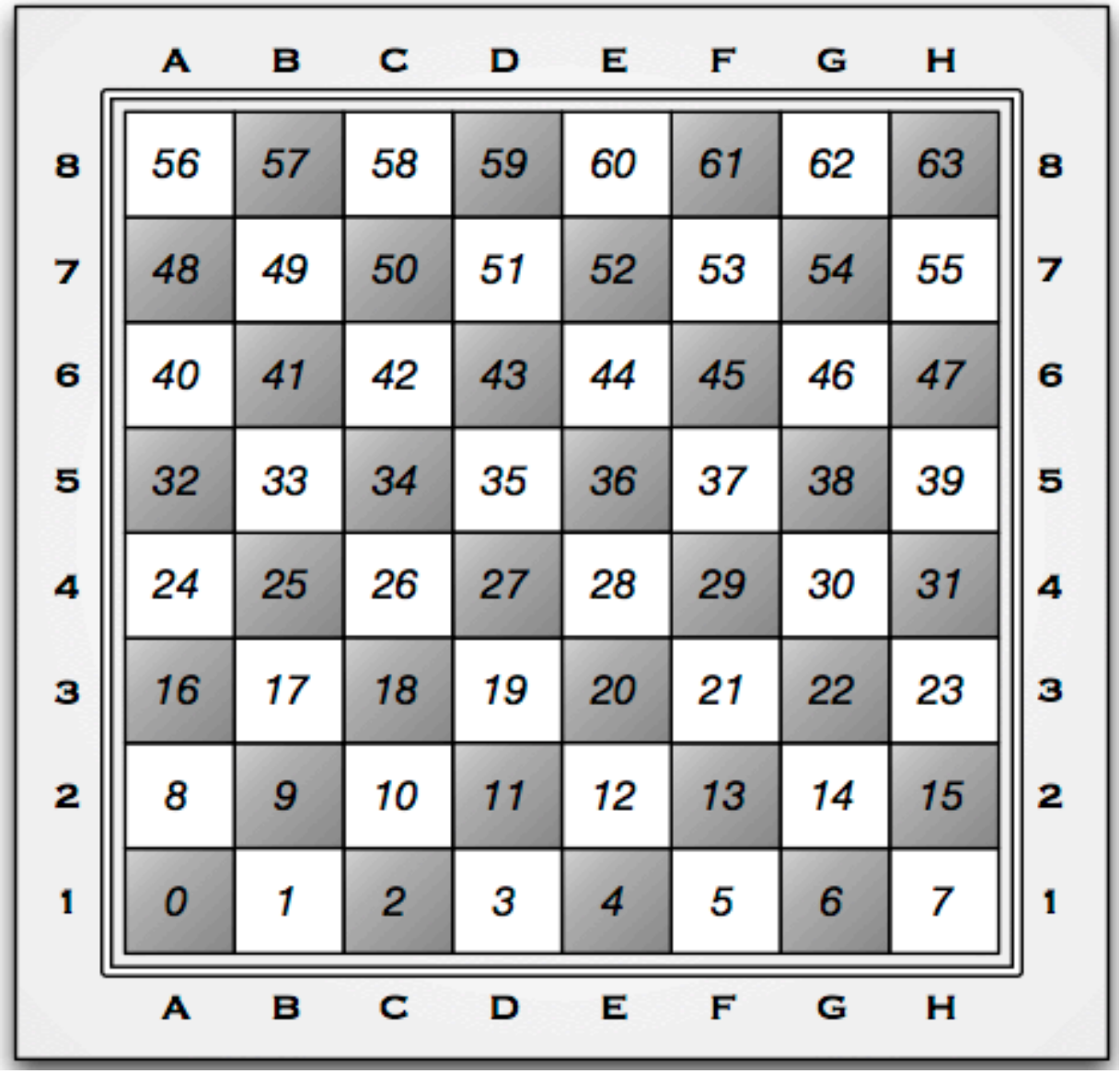
There are many different approaches to accomplish this. If we don't make an effort to think about an efficient approach the first that might come to our mind is using multi-dimensional arrays.
Soon this approach shows its inefficiency. Searching a simple element in the board implies to search - with a double *for* cycle - in all the arrays. So even if this approach is intuitive and might be easier than the one that we choose, it is not efficient enough to implement a serious chess player.

Because of our fixation in successfully achieve great results in terms of performance, we move then to another representation that use a concept called *bitmap*.

In this representation is used a sequence of 64-bit to represent every square of the chess table. We use an *unsigned long long int* to represent the chess table. This unsigned long long int variable[3] ends up converted in a 64-bit variable.

So this kind of variables let us make a correspondence between bit and table squares as shown in the table below.

[2] The common implementation of the heap region is a double linked list.
[3] From now on called bitmap.



GRAPHIC 1: CHESS BOARD REPRESENTATION

The numbers between 0 and 63 represent all the 64 bits of the bitmap variable. So the chess board A1 position is represented with the bit 0 of the bitmap variable and so on.

One of the major problems that we have with this representation is the fact that it is really complex, at first, to rapidly spot if the bit 0, for example, is the most significant bit of the bitmap variable or, on the contrary, the least significant bit. When we weren't used to this approach, we had a lot of headaches regarding this concept.

Just to mention it, as a base agreement we fixed the bit 0 – or the A1 square - to be the most significant bit of the bitmap variable. And, also, we fixed that between A1 square to H2, would always be placed the white pieces.

Representing the table and the pieces using a bitmap variable have the advantage to easily generate the movement of every piece using bitmask and shifting bits left or right. Another strong point of a bitmap is that is easy to retrieve information and generate additional bitmaps

by using a logical operator. For example an operation between two bitmaps can check if two pieces collide, and an or operation can give us the union of the two bitmaps.

In order to clarify to the reader the concepts reviewed here, below is attached the board implementation with the raw C code. We know that we have only to make an overlook of the board representation but we think that to fully understand how all works is necessary to have a fast reference code to have a general picture of the explained concepts. The raw code is also necessary to spot all that efficiency improvements done.

```
/*
 * This is a representation of the chess board. We can access the
 * field of this struct using the variable bitmap.
 */
typedef unsigned long long int BitMap;

/*
 * This struct is used when the movement generation is done.
 * Here we have all generated moves from one piece
 */
typedef struct _moveData {
        //A move generated from a particular board state
        BitMap generatedMove;
        //Contain the type of pieced killed in the move, or -1 if not
kill anyone
        int killedTypePiece;
} MoveData;

typedef struct Board
{
        //Arrays with all the BitMap of every white piece type
        BitMap whitePieces[N_TYPES];
        //Arrays with all the BitMap of every white piece type
        BitMap blackPieces[N_TYPES];
        //Structured with all the white, black pieced and the union
        //of the two
        BitMap totalWhitePieces, totalBlackPieces, occupiedSquares;
        //This variable identifies the generated moves of a piece type
        MoveData boardState[MAX_NUM_MOVES];
        enum piezas{
       //Enumeration used to assign a numeric value to every piece
type
                King =          0,
                Queens =        1,
                Rooks =         2,
                Bishops =       3,
                Knights =       4,
                Pawns =         5
        } tipo_pieza;
   //The color that is actually moving
   int actualColor : 8;
   //The color assigned to the player
   int myColor  : 8;
   //Heuristic value for this table configuration
   int heuristic;

}Board;
```

Given that we have already explained in one of the previous sections what every field is, here we will only review all that aspects that have not been explained yet.

On of the most important things that we have not already explained is that we work with groups of pieces instead of working with every piece independently. This means that for the explained variable **whitePieces** we don't have an array of sixteen - one per piece - bitmaps where we insert every piece position independently in the bitmap, instead we have N_TYPES of bitmaps where N_TYPES is the number of different types of pieces, such as rooks, pawns, knights, bishops, queen and king. So for example, in the pawns bitmap we insert the position of the initial eight pawns in the same 64-bit variable. So where a bit is set to one in the whole bitmap means that a pawn is occupying the correspondence board square.

Using this approach we can reduce the board representation size by 80 bytes. And even if it increases the complexity of the board representation is not something to care about.

As we have already explained, when you are used to, working with bitmaps is now a hard task and for strengthen this fact we can say that to know all the occupied squares in the board, we have only to make an or operation between **whitePieces** and **blackPieces** arrays, so it is not needed a double or triple *for* loop. The result of this operation would be stored in the **occupiedSquares** variable.

The next variable to be reviewed is the **boardState**. As can be seen, it is built with another structure called ***MoveData*** that is nothing more than a bitmap plus an integer variable. We have already explained its use, but we have not explained how we have dimensioned their contents. As can be seen, its length is given by the **MAX_NUM_MOVES** that is nothing more than the maximum total number of possible generated children for particular piece type. This number is calculated by identifying the maximum number of moves that a particular piece can perform and, eventually, all the maximum possible moves for all the pieces of the same type are added.

This variable helps us to keep track of the generated children, but it's not the total number of generated children because of the magnitude of this number. The board representation is something that must remain as smaller as possible so we only store the children generated by each piece type and we store all the possible generated children in a temporal variable that is not directly related to the board representation and is only used in the search algorithm.

In order to optimize the board representation size we declared the **actualColor** and **myColor** variables as integers variables but only of 8 bits instead of their normal size[4]. As we know that their values could only be 0 or 1, we only need two bits[5] to store them. But even we only need two bits to store them we know that the compiler doesn't store just two bits in memory because the processor have to work with sizes multiple of 8; the processor works, at least, at byte level. So we save the time used by the compiler or the processor - we are not sure about who perform this task – by setting the variable size to eight bits.

Finally, we have used an enumeration to specify the piece type number assigned to every piece type so we didn't have to remember the exact numbers to access the correct bitmaps in the white or black pieces arrays and in many other situations.

---

[4] In most 32 bits architectures the size of an integer variable is four bytes.

[5] We need two bits to identify the actual value – one or zero – and the extra bit to identify the sign of the integer value.

The creation of this enumeration was of major importance given that this helped us to use a standard to access and create all the structures and algorithms with a given order. This fact is difficult to explain without seeing the particular code where we used it, but to know its importance only remains to say that this enumeration is used in almost all the pieces of code.

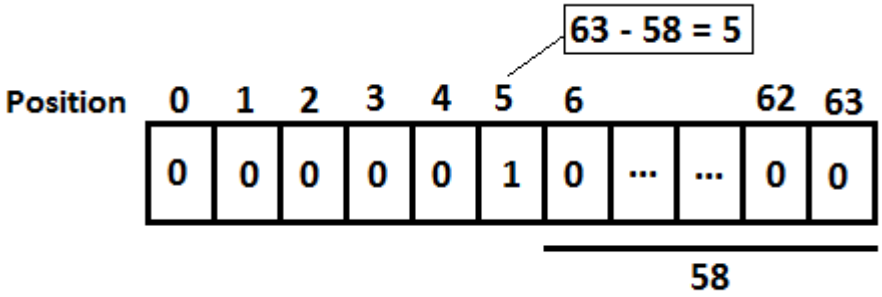
To conclude, with all our efforts to improve the space efficiency of the board representation and therefore the speed of the whole program we ended up with a board representation of only 454 bytes approximately, less than a half KB. We are really proud of this result and we thing that this is one of the key factor that let us so great results in the performance tests.

MOVE GENERATION

The generation of the movement for every type of pieces is made using the binary operators. For every piece type there's a function that calculate all the possible movement; moreover during the move generation the collision with other pieces on the board are detected and managed.

The first step in the move generation is retrieve the ***position*** of a piece on the board: the *position* is a numerical value that indicate what's the bit switched to 1 for that piece; for example if we have a pawn on the table in the square 2-C, the position is 10.

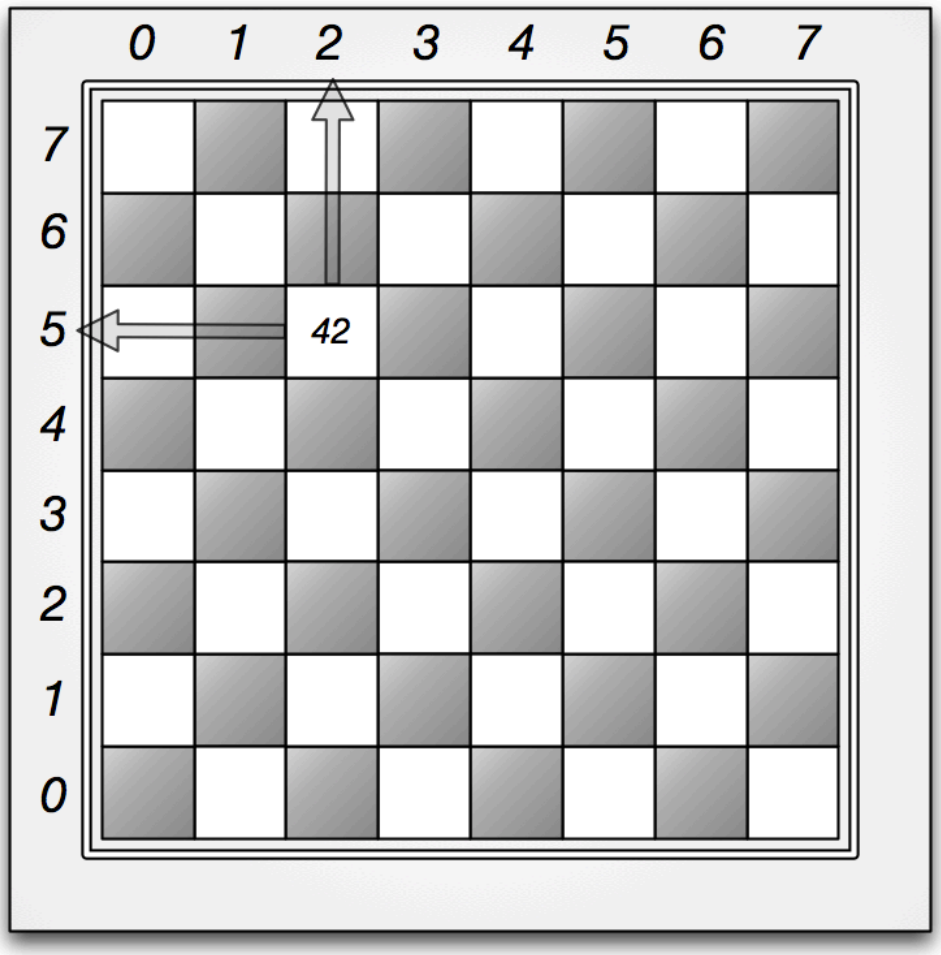
To calculate the position of a piece in the chess we basically do what explained in the next image:



We shift the bitmap, and count how many shift are needed to put the bit set to 1 in the last position, and then we calculate the difference 63-*#shift*, and obtain the position of the bit in the bitboard, that reflect the position of the piece in our board representation.

For the generation of the movement we need also the row and the column where is positioned the piece. To calculate this we make:

- **Row**: position%8
- **Column**: position/8



We enumerate the rows and the columns within the range [0,8], in the way shown in the image.

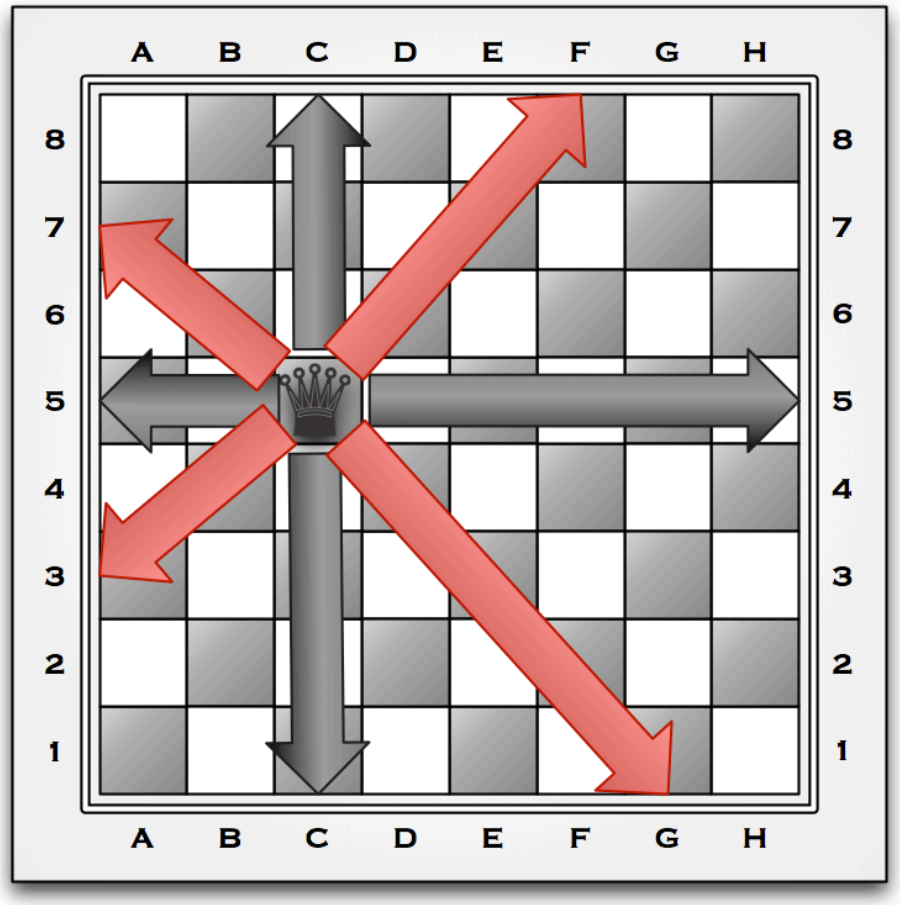
Once we have calculated the position, the row and the column, we can calculate the movement for every piece type; next in the document we explain the movement for every piece type.

We also explain how we store the movements and how we store additional information that can come out from the move generation, as information about the killing that happen during the movement of a piece.

All the images that we insert in this document, that represents a chess table, are from white perspective; we explain how we generate movements for all the piece type from the white point of view. The generation of move for the black pieces is the same for all the piece type, excluding the pawn, which we analyse detailed later.

QUEEN MOVEMENTS

The queen is the piece that has the maximum possibilities of movement. In the following image is shown the movement of a queen:



As we can see there are two different type of movement: a diagonal and a straight movement, so we have split the functions that generates the movements in two: one function to generate the straight and one for the diagonal.

Let's start explaining how we make the straight movements: to move the queen left or right, we have to calculate, using the *column*, how many movements we can make left and right. Then we make all this horizontal movement by shifting left or right for the number of times calculated before.

To move the queen up and down we calculate, as before, how many time we can move up and down, by using the *row*; once calculated the movement, we have to shift to the left 8 times to move the queen one square down, and shift to the right 8 times to move one square up. As before, we repeat this movement as many times as calculated to stay in the board bound.

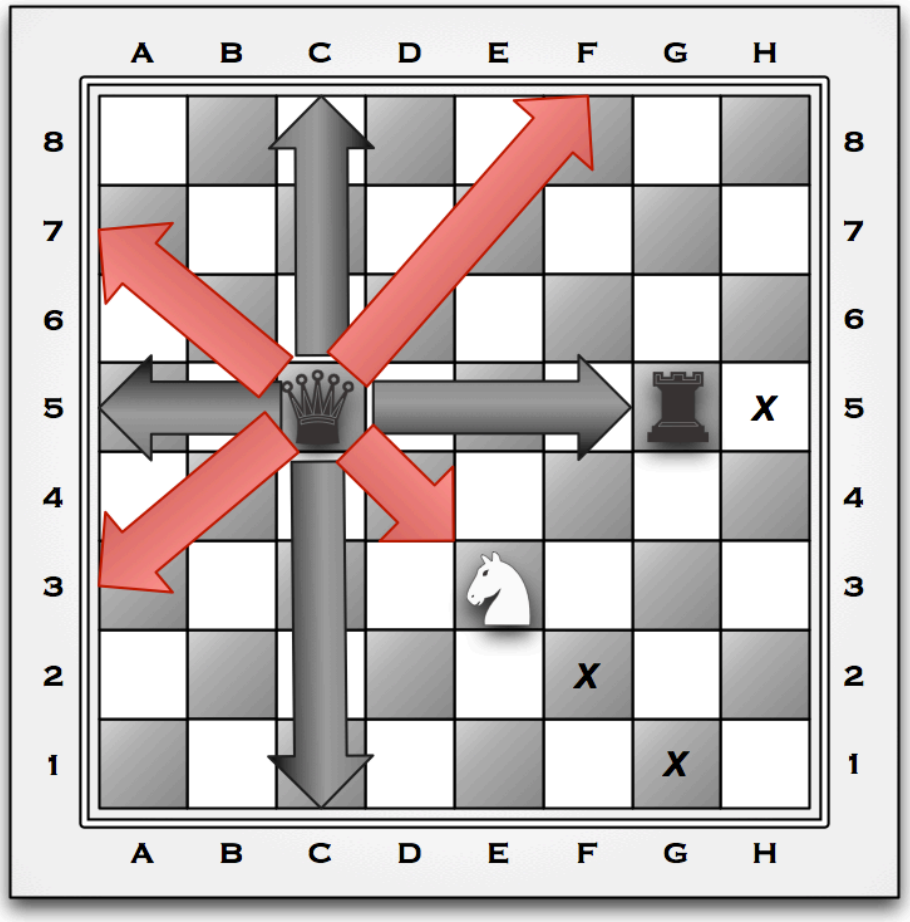
To generate the long diagonal move we have to calculate, using the *row* and the *column*, how many movement we can do up-left, up right, down-left and down-right. Once calculated that, we use another time the shift operator: to make the queen move up-left we have to shift 7 time, and to make the queen move up-right we have to shift 9 time; we can see this

movement as a variation of the up movement, because it's like moving the queen up and then move to left and right.

The same things happens when we make the movements down-left and down-right, what change is the direction of the shift: to move down-left we have to shift left 9 time and to move down-right we shift left 7 time.

Every single movement make generate a bitboard, that represent the new position of the queen, and every single movement is saved to be used in the future.

Once we created the function that generate the movements we have to check if every single movement is possible: let's think what happen if we want to move and there are other pieces on the queen's way, like shown in the next image.



In this image we can see that the queen movements are limited by two different pieces: the down-left movement is limited by the presence of a white knight (a piece of different colour), and the left straight movement is limited by the presence of a black rook.

The first thing to do in this case is stop generating other movement in the collision direction, because clearly the queen cannot make other movement in this direction; every time that we generate a movement we check if this movement is in conflict with other pieces in the board. This is an easy operation to do using bitmap, because we make the binary and (*&*) between the generated movement and the bitmap *occupiedSquares* that we have in the *board* structure. If

the result of this operation is different from 0, we have found a conflict, because it means that there're two bit set to 1 in the same position in both the bitboard; we can now stop the generation of movements in the direction of collision.

One additional thing that we do in case that a collision is found, is check the type of this collision: as we can see if we have a collision with a piece of our colour (as happen in the image between the black queen and the black rook), we cannot occupy that square and we have only to stop the move generation.

If the collision is with a piece of the other colour (as happen in the image between the black queen and the white knight), we have found that our piece can kill another piece, and occupy the square of collision. In this case we've create a function that resolves that conflict: the function checks the colour of the piece that make a collision by applying the binary and operator between the generated movement and the bitmap *totalWhitePieces* or *totalBlackPieces*, according to the colour assigned to us.

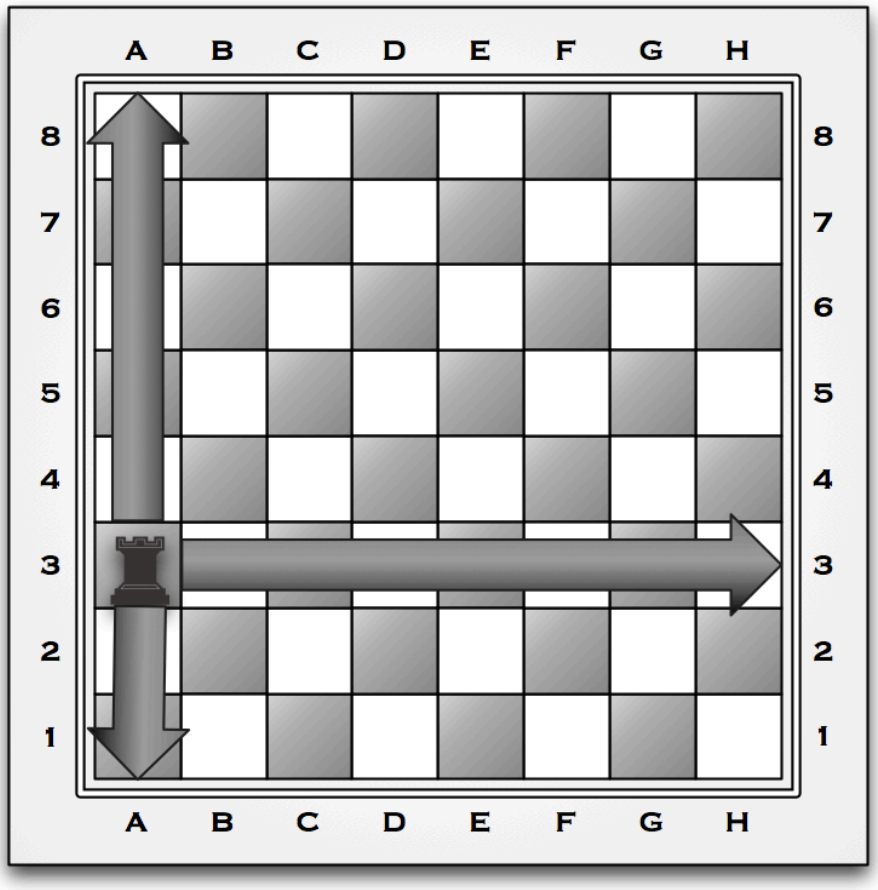
Once found that the collision is with a piece of another colour, we can make additional operation, to add extra information to the generated movement. When there's a collision with a piece of the opposite colour, means that we're killing this piece: we can save this information and store for later processing.

When we found that our queen kill an enemies piece, we found and resolve the conflict, but we can make more; what we do is looking in all the bitboard of the opponent colour and search for the position where we found a killing conflict: in this way we can calculate what type of piece is killed by the movement and where it's positioned. We make another time the logic and between the movement and all the bitboard of all the pieces type of the opponent colour.

At the end of these process every movement is saved inside a C-struct that contains the generated movement and an information about what type of piece is killed by making this movement, if there's no a killed piece for a movement the last value is set to a negative one.

ROOK MOVEMENTS

The movements of the rook are only straight movement, for an arbitrary number of squares, as described in the next image.

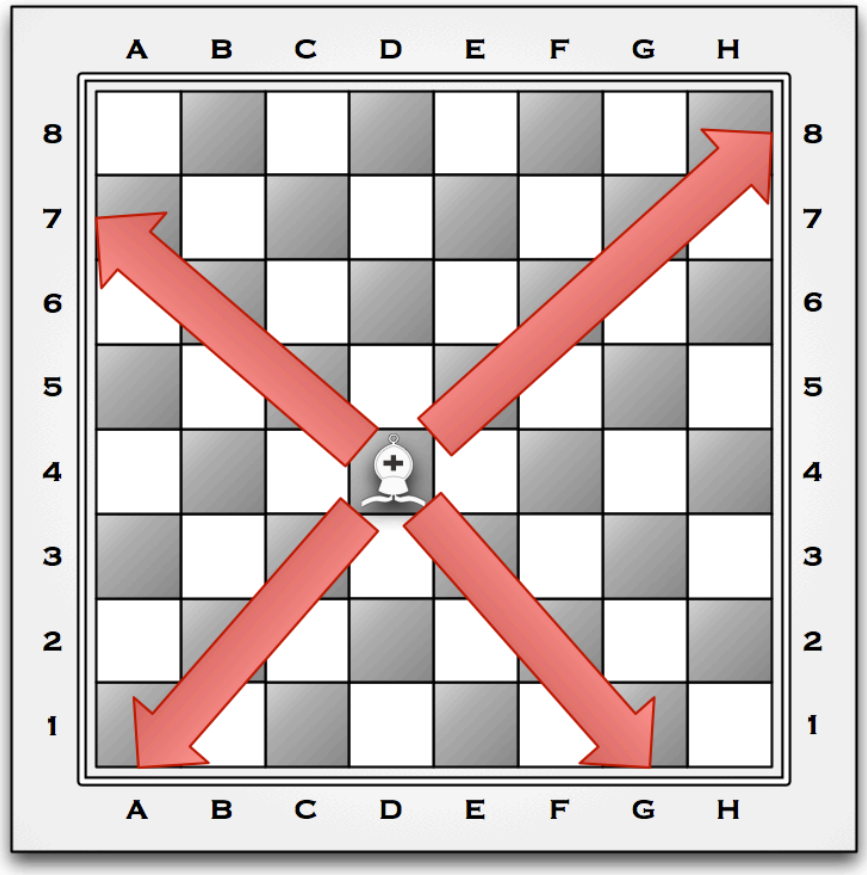


The function that generates the movements is the same that generates the straight movement for the queen: shift left or right by 8 to move up and down, and a simple shift of one to move left and right; all the shifts are made within the boundaries of the chess board, calculated using the information *row* and *column*.

Like the movements generated for the queen, also for these movement (and for all the movement generated for every piece) we check if it is in conflict and the type of conflict. The check of the conflict is made as explained before, and permit to stop generating movement if in a given direction is found a conflict, and check if a conflict kills some type of piece; then all the struct representing the movement are created and saved for future processing.

BISHOP MOVEMENTS

The bishop can move only diagonal and, like the queen and the rook, cannot “jump” if there’s a piece on his way. The movement may have arbitrary length. The next image shows the possible movements of a bishop.



To generate the movements for this type of piece, we use the function that generates diagonal repeated move, as made for the queen’s diagonal movements: we shift 7 or 9 times, to the right and to the left to generate al the diagonal movements. Like we do with the queen we use the *row* and *column* information to give a limit to the generation of movements.

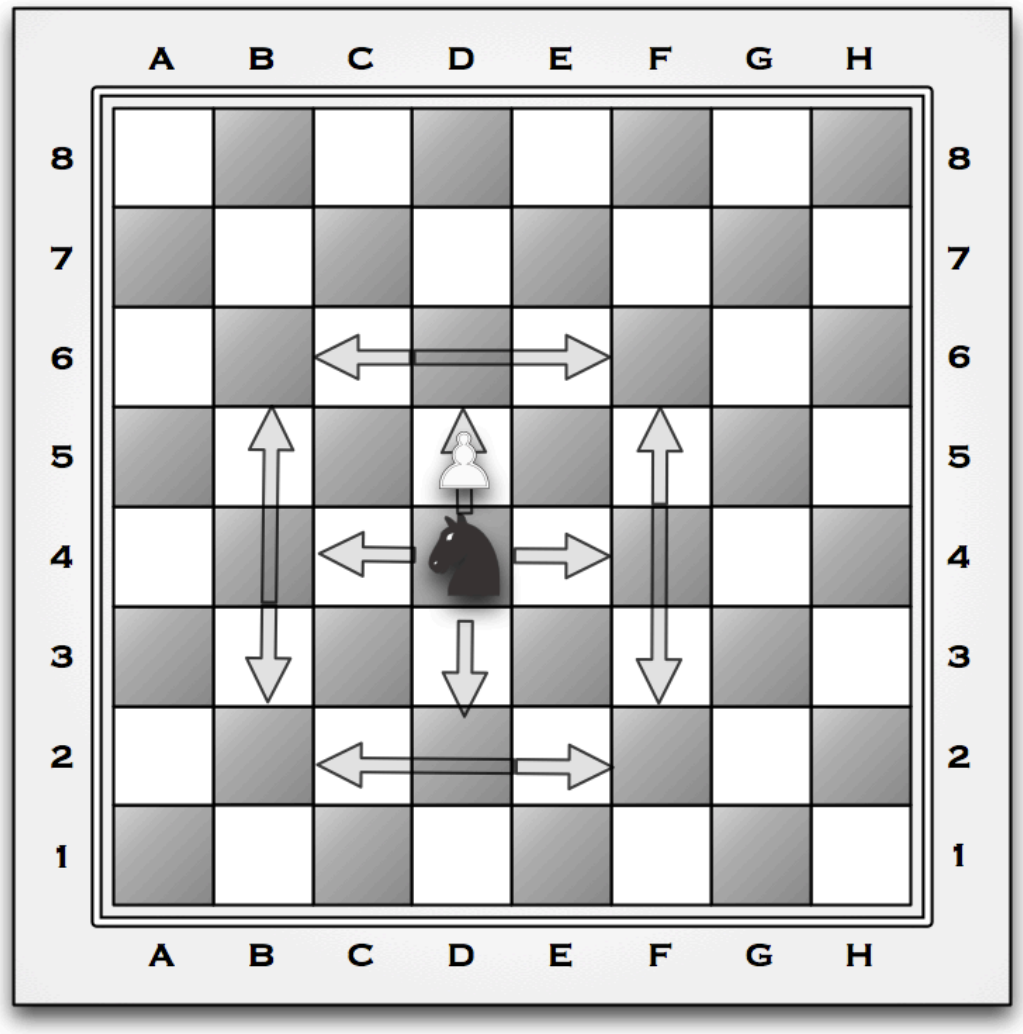
Like the queen and the rooks, also for the bishop’s movement we check if there’s a conflict to interrupt the generation of movement in a determinate direction, and we calculate also if there’s a kill that comes from a conflict.

At the end of this process all the structs that contains movements and killed type pieces are generated and saved.

KNIGHT MOVEMENTS

The movements of the knight are different from the ones seen before: the knight has to make a movement of two square in a straight direction, and then has to move one square more in a direction perpendicular to the previous.

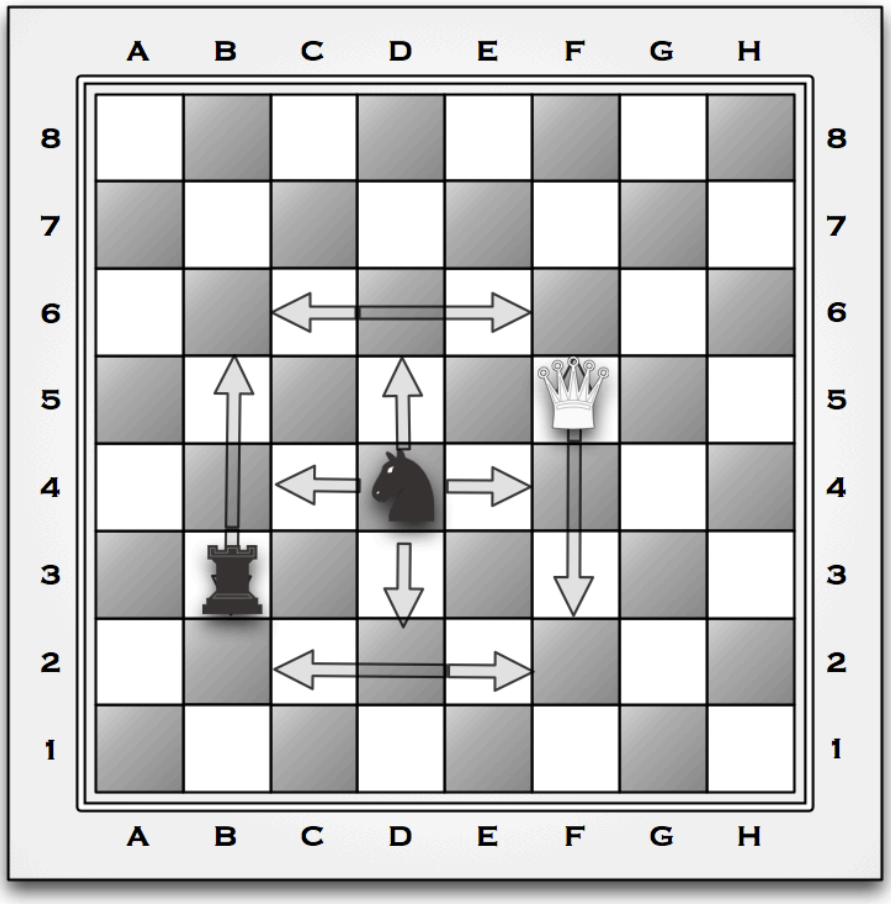
The knight can also “jump” if there’s a piece on his way, as described in the following image.



The generation of the movements for this type of piece is made, as the other before, using the binary operator shift: to move two squares up and one left we make a shift right of 15 position, to move two squares up and one right, we make a shift right of 17 position.

We can see these movements as a concatenation of two moves up or down, which are two shift of 8 position, and a movement left or right, that is a shift left or right of one position. Before making all these movements, we check if it’s possible and do not go out from the boundary of the chessboard, using, as usual, the information *row* and *column*.

In the generation of these movements we do not check if there’s a conflict during the generation, as we made before, because the knight can “jump” if find a piece on his way. What we have to check is the presence of a conflict in the square where the knight is moving, as shown in the next image.



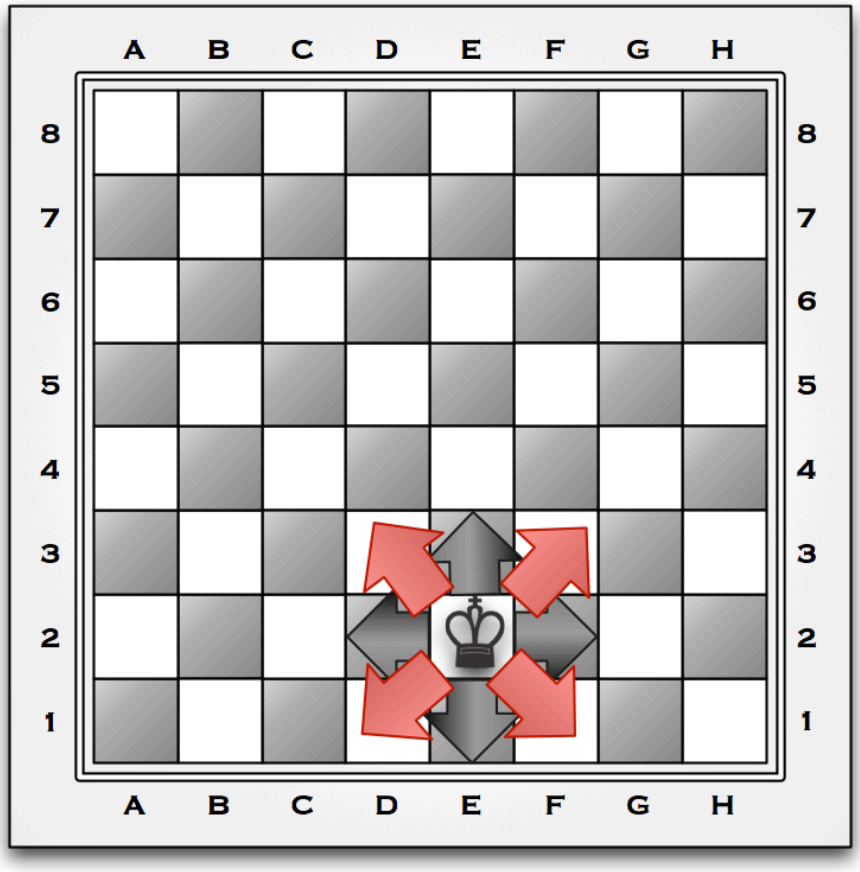
In this case in the square 3-B we have a conflict with a piece of our colour and in the square 5-F a conflict with a piece of the enemy's colour.

What we do in this case is checking for the conflict as before, by applying the and operator to the movement's bitboard and the *occupiedSquares* bitboard. If the conflict is with a piece of our colour, we cannot occupy the conflicting square, so the movement are trashed and do not saved for future analysis; if the conflict found is with a piece of the enemy's colour, we have found a kill movement, and we check the type of piece killed in the movement (in the example is the white queen).

Once we've retrieved this information, we can create all the structs with the type of piece killed and the movements, and save it for later use; logically, also movements that do not kill are saved and stored.

KING MOVEMENT

The king is a piece that can move in any direction for only one square, as shown in the next image.



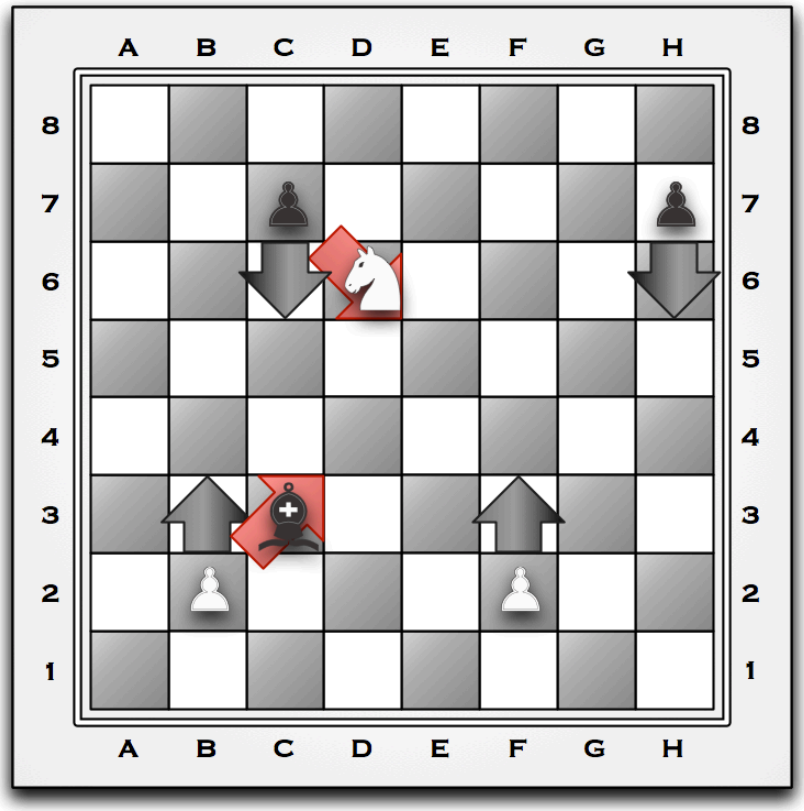
The generation of movement are very simple: using the shift operator we generate the movement as usual (8 shift left and right to go up and down, and shift of 1 to move in the right or left direction).

Like we do for the knight, we have to check for the conflict in the destination square, and we manage the case of a killing move as usual, by saving it for later usage.

PAWN MOVEMENTS

The pawn is a piece that can move only forward, and can move for two squares if it moves from the start row; in all the other situations can move for only one square.

The difference between the pawns and the other type of pieces is that the pawns uses one direction to move, but can kill only in the diagonal direction, like the bishop; also in this case it can move only one square. The movements for the pawn, including the killing movement, are shown in the next image.



All the moves generation functions seen so far are colour-independent, because a piece can move both forward and backward; the generation of movements for the pawn are colour-dependent, because the pawn can only advance.

The generation of a movement are, as usual, a shift of 8 squares, and the shift is right if the pawn is white (8 shift right makes the pawn go up), and left if the pawn is black (8 shift left makes the pawn go down); we, as usually, check if a movement is inside the table bound by using the information *row*. Once we've made this movement and save for later use, we check if the pawn can kill: we generate two movement, respectively up left and right for the white pawns, and down left and right for the black pawns. We examine these movements by making the logic and between the movements and the *totalBlackPieces* or *totalWhitePieces*, according to the enemy's colour of the pawn. If we found a killing collision, we retrieve the type of piece killed and save the movement; instead the movement are discarded.

CHILDREN GENERATION

Now that the move generation is explained we can pass to explain how we have generated the children used in the search algorithm.

The first thing that must be emphasized is that we have avoided the use of dynamic memory inside the search algorithm. Given that for the operating system is much more complex to work with the heap memory region, allocating and freeing variables, we decided to store all data in the stack region. This, of course, limited our algorithmic options and our space optimization, but thanks to this decision we really speeded up the search algorithm.

So as we've said, in the search algorithm functions we don't allocate nor free dynamic variables in execution time, besides, we create a complete array of possible generated children in compiling time.

This approach means that we have to know beforehand the maximum number of children given a parent or, what is the same, given a particular game state. One could incorrectly think that the number of maximum children given any root node is not affordable, but even if this is an intuitive thought it is not the correct thought.

In order to calculate the maximum number of children that any node could generate, we only add the maximum number of possible movements that each piece can perform starting from the square in which it can perform its maximum number of moves. This position is always, more or less, the center of the board. So, for example, the maximum number of moves – and therefore, the maximum generated children – that the queen can perform is 27. We add all these values for every piece and ended up having an array of 137 positions.

```
Board children[MAX_CHILDREN];       //Array of children
```

Once defined the variable in which we would store all the children, we thought how these children would be generated.

Thanks to our modulated code, perform the children generation was not a hard task. The only thing we had to do was to generate all piece movements as explained in the previous section. And eventually we store all that generated movements in the children array.

SEARCH ALGORITHMS

In this project we have implemented and tested two different search algorithm: the negascout and the negamax algorithm. Both based on the minimax approach for the search in game against enemies, but use different approach to search. Next we make a short introduction of both the algorithms and its features and then explain how we implement them, test and choose what we think is the best for our software implementation.

NEGASCOUT

Negascout is a search algorithm proposed by Alexander Reinefeld in 1983, mainly used in games. It's an algorithm from the family that allows search against opponents and it's a combination of two different search algorithm, the Scout and the Pruning alpha--beta algorithms.

Scout algorithms is a search algorithm based on a *Scout* function: this function is a boolean one used to analyze a sub--tree of the search tree, and evaluate if it's necessary or not to search in it. This algorithm is based on the assumption that, if we know a value *v,* it's easier evaluate if a subtree has a value greater or smaller than *v,* rather than calculate the exact value of that subtree. So the boolean function is used to evaluate if we must search in a branch or not, and the algorithm go straight to the solution.

The other algorithm, alpha-beta pruning, used a depth-first like search to find a value for the variables α and β, and evaluate, depending on the value of these variable and on the presence of a node MIN or MAX, if we can prune a node and all of his son. The alpha-beta algorithm speeds up the search by pruning irrelevant branches of the subtree without loss of information.

It is possible to modify alpha-beta to have a behavior similar to the Scout algorithm using the technique of Null-Window search: given a value m, we scan the tree using the alpha- beta algorithm whit m and m+1 as value for alpha ad beta. In this case the algorithm can fall-high (means that it return a value that is larger than it's upper bound beta) or fall down(the same as before, with a value less than lower bound alpha). We can use this technique to say if a branch has a value less or more than m, scanning it with a Null- Window search and evaluating the value returned. This behavior is equivalent to the boolean evaluation function (the Scout) of the Scout algorithm.

With these elaboration, Reinfield create the Negascout algorithm: NegaScout just searches the first left move (and all of his left child) with an open window (α and β are --∞ and +∞), and then every right move with a null--window. Negascout, in fact, "scouts" the successor of a node (in the first iteration is the root) from left to right; the leftmost son is searched using a infinite window, and the other are searched using the null-window.

If the null windows search falls--high (we find a bigger value), we have to search the same sub-tree with a wider window.

Here above there's the algorithm in C-style code:

```
    int NegaScout ( position p; int alpha, beta )
1.  {                       /* compute minimax value of position
    p */
2.    int a, b, t, i;
3.    if ( d == maxdepth )
4.       return Evaluate(p);                          /* leaf
    node */
5.    determine successors p_1,...,p_w of p;
6.    a = alpha;
7.    b = beta;
8.    for ( i = 1; i <= w; i++ ) {
9.       t = -NegaScout ( p_i, -b, -a );
10.      if ( (t > a) && (t < beta) && (i > 1) && (d <
    maxdepth-1) )
11.         a = -NegaScout ( p_i, -beta, -t );      /* re-
    search */
12.      a = max( a, t );
13.      if ( a >= beta )
14.         return a;                                  /* cut-
    off */
15.      b = a + 1;                        /* set new null
    window */
16.   }
17.   return a;
18. }
```

The variable *d* indicate the actual deepening in the tree, as we can see in the algorithm at line 3. *Evaluate* is a function that return the actual value of a leaf node. The algorithm is recursively invoked on all the left child, until we find a value *t;* we use this value to set a null-window and scan all the right child with this window. As we can see in line 10, if the value of one the left child is bigger than the actual leftmost child value *a,* or less than the beta value, we re-search the sub-tree with a window wider (--∞, t).

The seeing of the null-window happen in line 12 and 15: we find for the max value between the find *t* and *a,* and *b* is set as *a+1,* and then the for cycle use this window for all the right brother of the node.

The cut-off (as in Pruning alpha--beta) happen at line 14: if a find value is greater than the value *beta* there's no need to search further and the node was cut. This algorith use a *NegaMax* approach, that we explain better in the next algorithm, to use only one function to evaluate if the current node is better that the previous.

The performance of these algorithm are good on tree with a branching factor from 20 to 60, and the performance are even better if there's some sorting of the child, localizing the best children in the leftmost part of the search tree.

NEGAMAX

The negamax is not proper a search algorithm, it's a way to implement minimax and all the derivated algorithms. Instead of using two separate subroutines for the Min player and the Max player, it passes on the negated score due to following mathematical relation:

max(a, b) == -min(-a, -b)

What we do when implementing negamax is using a classic minimax algorith. The version of negamax implementedi in the software use minimax and add the alpha-beta pruning. The code of the algorithm (C-style code) is the following:

```
int Negamax_alphabeta(position p, int d, int alpha, int beta)
1. {
2.       if (d == 0) return eval();
3.      determine successors p_1,...,p_w of p;
4.       for (i = 1; i < w; i++)
5.       {
6.              val = -NegaMax_alphabeta(p_i, depth-1, -beta,
   -alpha);
7. // note the minus signs & switching of places of alpha and
   beta
8.              if (val >= beta) return beta;
9.              if (val > alpha) alpha = val;      // both
   sides maximize
10.      }
11.      return alpha;
12. }
```

As the algorithm before let's analyze the code. If we reach a leaf node, we call the function *evaluate()*, that return the value of the heuristic for a leaf node, as seen in the line 2.

In the for cycle between line 4 and 10 we call the NegaMax on all the child node generated in line 3; let's pay attention to the fact that the value returned from the invocation is negated and in the invocation (at line 6) the value alpha and beta are negated and switched. In line 8 we can find the pruning, that permit to do not search in all the subtree and returning immediately a value. In line 9 we found the maximization function that permit to store in the variable alpha the maximum of the child at a given level and return it at the end of the algorithm at line 12.

A note for both the algorithm presented: in order to make work properly the *NegaMax* approach, the evaluation function has to return a value that has a sign positive or negative respecting the side to move: if it's a maximizing function (we are moving) the sign willl be negative and the opposite if is moving the adversary. This point will be explained well when we talk about the heuristic function.

IMPLEMENTATION AND IMPROVEMENT

The first algorithm to be coded was the NegaScout (all the algorithm can be found in the file called search_functions). We basically generate all the child of a given node: the child are all the movement generated from a board configuration, and then recursively call the algorithm on every node.

We choose this algorithm at the beginning because it work well with tree with an elevate branch factor, as in our case with the chess. Once coded we test it with different board condition and with different depth on different machine. What we found is that at the initial and in the middle game situation, the number of the child is elevate: let's think that the maximum number of child is 137, almost the double of 60, that's th condition for the NegaScout to give improvement over alpha-beta. Moreover, create a children sorting function seems to be unproductive, we risk to spend more time on the child ordering than searching. So we decide to move to the classical NegaMax approach and add some extra feature to tuning it up for the chess search.

The first implementation of NegaMax alpha-beta algorithm is the one explained before: with this implementation we can see that all the time of the machine was spent evaluating and searching and the pruning come more frequently; this is given to the fact that the NegaMax does not have the null-window search that characterize the negaScout.
Thinking about the search problem we have figured out that we can adapt the search to the chess game; let's assume that we are searching with a depth of 5, what will happen if in the second depth we make or we suffer a check? The search will continue until level 5, and we are generating all the child of the node where there's a check; all of this is unecessary and is a waste of resource and time.

What we have done is create a function that evaluate if in a previous node there's a check: this function look in the *board* data structure and look at the bitboard that represent our king and the enemies king.

If the bitboard of our king is set to 0, means that in the previous node we're in check, so we stop the recursion and return a high negative value to indicate that in on this path we lost the king.
If the bitboard of the enemies king is 0, means that in the previous node we've made a check, and we stop the recursion and return a high positive value, to indicate that this is a good path. These evaluation do not substitute the evaluation function, based on the heuristic that we'll describe later in the document, but it's a way to avoid searching in branch where there's not the our or enemies's king.

The modificated version of the algorithm is the following :

```
int Negamax_alphabeta(position p, int d, int alpha, int beta)
1. {
2.       if (d == 0) return eval();
3.      determine successors p_1,...,p_w of p;
4.       for (i = 1; i < w; i++)
5.       {
6.              if( evaluate_our_check == KING_IN_CHECK) //our
   check
7.                  val = OUR_CHECK * sideToMove(d);
8.              if (evaluate_enemies_check ==
   KING_IN_CHECK)/enemies check
9.                  val = ENEMIES_CHECK * sideToMove(d);
10.             val = -NegaMax_alphabeta(p_i, depth-1, -beta,
   -alpha);
11.             if (val >= beta) return beta;
12.             if (val > alpha) alpha = val;
13.      }
14.      return alpha;
15. }
```

In the lines between 6 and 9 we evaluate first our check and then the enemies check; the condition are mutual, we can found our check or a enemies check.

The function *sideToMove()* is used to make the NegaMax approach to work properly, it takes as parameter the current depth in the search tree and return the correct sign for the value to be returned.

The real implementation of the code is almost similar to the one proposed here above; the main difference is that in the “ideal” algorithm, the best heuristic value are propagated to the upper level of the tree and then returned to the root, but in the real implementation we need a way to track the path between the best leaf node and the root.

We solve this problem using this strategies: we accept as a parameter a reference to an empty bitboard, where we store the value of the next move to do, the first node on the path between the root and the best leaf node.

We call the algorithm as usual, and the return value of the algorithm is an integer, but we add a control that permit to store the best child of the first level, that is the best next move to do. If we are in the depth 0, that is the first call of the algorithm, every time that we change the alpha value (line 12), we store the bitboard that have this value.

INTRODUCTION OF THE CHECKMATE

Using the chess program we note that everytime that the machine can make a checkmate, it prefer to cover the king or make move that induce the enemies in check, but difficultly it individuate that can close the game with a check mate.

To solve this problem we've create a function that is called inside the search function, just after the point where we obtain the value of *val* (line 10).

The function, in this document we'll call it *checkMate()*, check if our opponent is in checkmate and add a "bonus" at the value *val*.

To determine if the adversary is in checkmate, we check if the enemie's king is menaced; if it is we check if it is menaced in all the movements that can do. If this condition is verified, we have found a checkmate.

HEURISTIC FUNCTIONS

The heuristic functions that we are going to describe are used to evaluate a leaf node in the search tree; we use different heuristic function to evaluate the features of the game board that we are evaluating and then we combine the result of every function. All of this operation happen when is called the function *evaluate().*

To evaluate a particular board situation we use different type of heuristic:

- Material Heuristic
- Position Heuristic
- Menaced Heuristic
- Check Heuristic

Next in the document we explain how these different function works and how we combine these to find a correct value for a node.

To understand how we have integrate the heuristics functions in the program we need to explain also the concept of ***game phase.*** We divided the match in three different part:

Opening: the initial phase of the game, where almost all the pieces are on the board. We fix the bound of these phase to 8 move.

Middle game: the phase of the game where some pieces are dead, most of them are pawns, and where the two player try to check one each other.

The transition between the opening and this phase is done when 8 movements are made. We remain in this phase until the material heuristic (see after) does not reach a determinate value.

End game: the final phase of the game, where only few pieces remained on the board and where the check are more frequently and the probability to have a checkmate is elevate. We enter in this phase when the material heuristic of our pieces fall below a fixed value.

We vary the combination of the different heuristics and also the heuristic, according to the current game phase.

To calculate the current game phase we maintain a variable that store the number of current move; when we call the *evaluate()* function, we calculate the current movement, according to the depth of the game tree.

MATERIAL HEURISTIC

This heuristic, as the name suggest, calculate how much “material” is positioned on the chess board. We assign to every type of pieces a material value:

Queen: 900

Rook: 500

Bishop: 325

Knight: 300

Pawn: 100

To evaluate the quantity of material present in the board for a single color, we first have to calculate the number of pieces of every type.

First created a function that count the number of pieces in a bitboard: we shitf the bitboard to the right for 64 times, and every time we make the logic and between the shifted bitmap and a bitmap that has a bit set to one in the last position; if the result is different from 0, we have found a piece. We pass to this function all the bitmap that represent all the type of pieces; once counted the pieces we multiply the number of piece found for the weight of every piece type, according to the following formula:

Material_white = #white_queen * queen_weight + **#white_rooks** * rook_weight + **#white_bishop** * bishop_weight + **#white_knight** * knight_weight + **#white_pawns** * pawn_weight

The same operation is done for the all black pieces. Note that the king is not counted in this heuristic, this is because we check the presence of the king in another heuristic function and also in the search function, because the king is the most important piece. Once we calculate the value ***Material_Black*** and the value ***Material_White***, we calculate the final value of the heuristic; we check the color assigned to us and, according to our color, we made this operation:

We are **white: Material= Material_White - Material_Black**.

We are **black: Material= Material_Black - Material_White.**

In fact we are calculating the difference between the material that we own on the board and the material owned by the adversary: if the value calculated is negative, means that our adversary has a material advantage, if it is positive, we are advantaged.

POSITION HEURISTIC

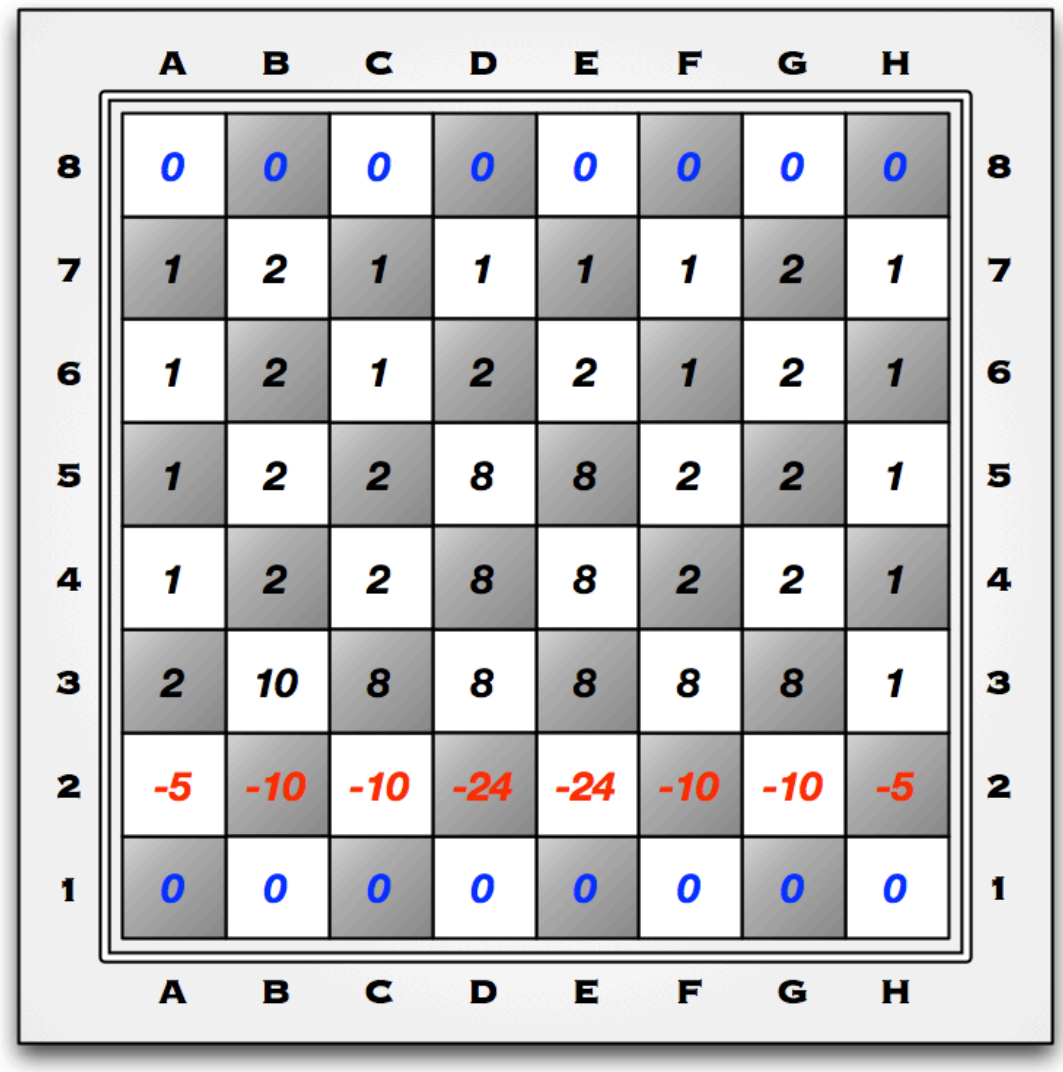
This heuristic is based on the idea that, in a determinate game phase, a position is better than another, for a determinate piece. For example the king in the initial phase does not have to move around, but stay covered in the rear.

This function also vary with the game phase, because it is safe that for the king in the opening and in the middle game to stay covered, but in the end game the king may be moved to other places to being saved.

To calculate all of these value we have created, for every piece type, a representation of the chess table, where we put the value of this heuristic. Next we pose all the heuristic position table that we use and explain the mean of each. Note that every table is from white point of view, we explain next how we adapt it to work also with the black pieces.

PAWNS POSITION

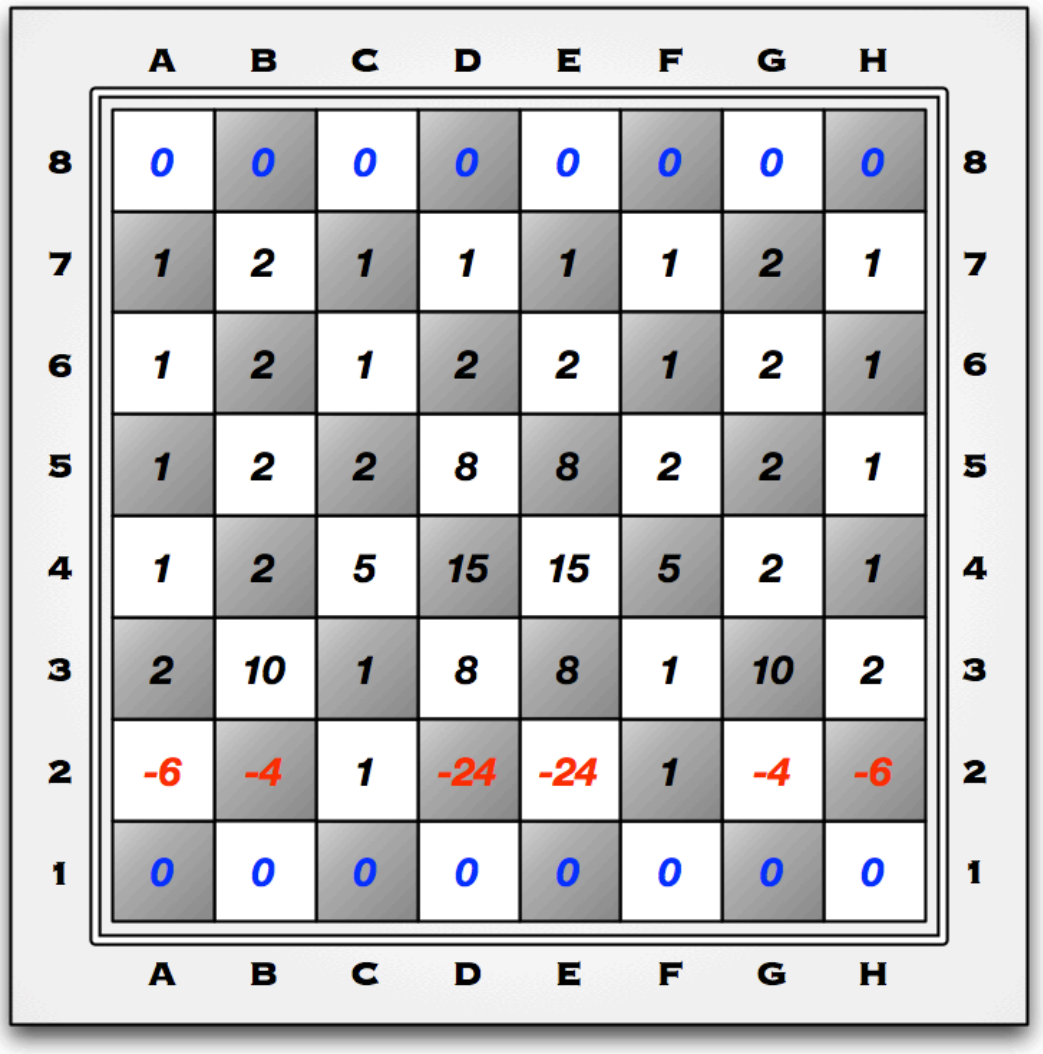
OPENING GAME



In this figure we can see the value of the position heuristic for the pawns in the opening game: are enlighted in red the squares with a negative value, in blue the neutral, and in black th ones with a positive value.

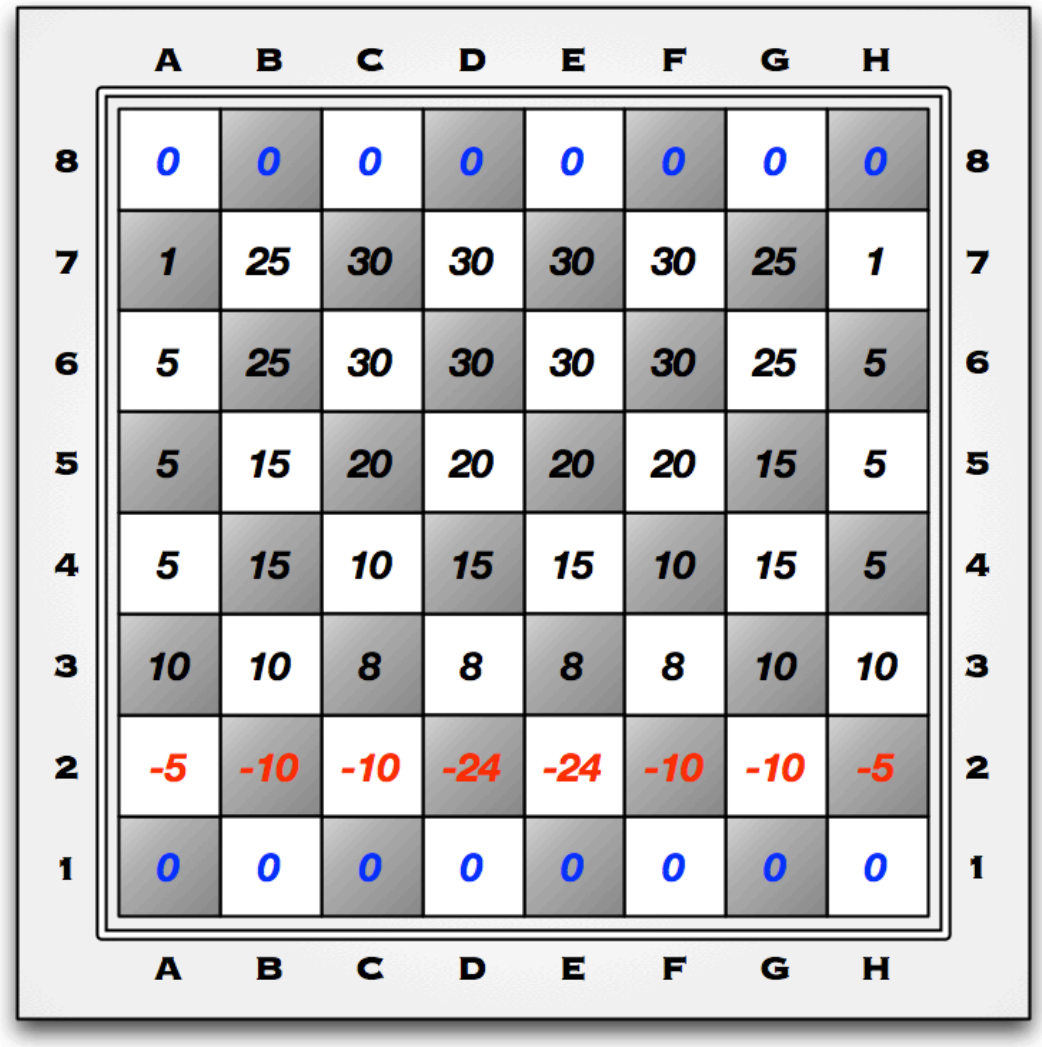
In this phase of game the pawns are encouraged to go ahead, and not to stay in the second row that have a negative score.

MIDDLE GAME



In the middle game the pawns are encouraged to go more forward and we made small change, according to the behaviour that the chess engine demonstrate in game.

END GAME



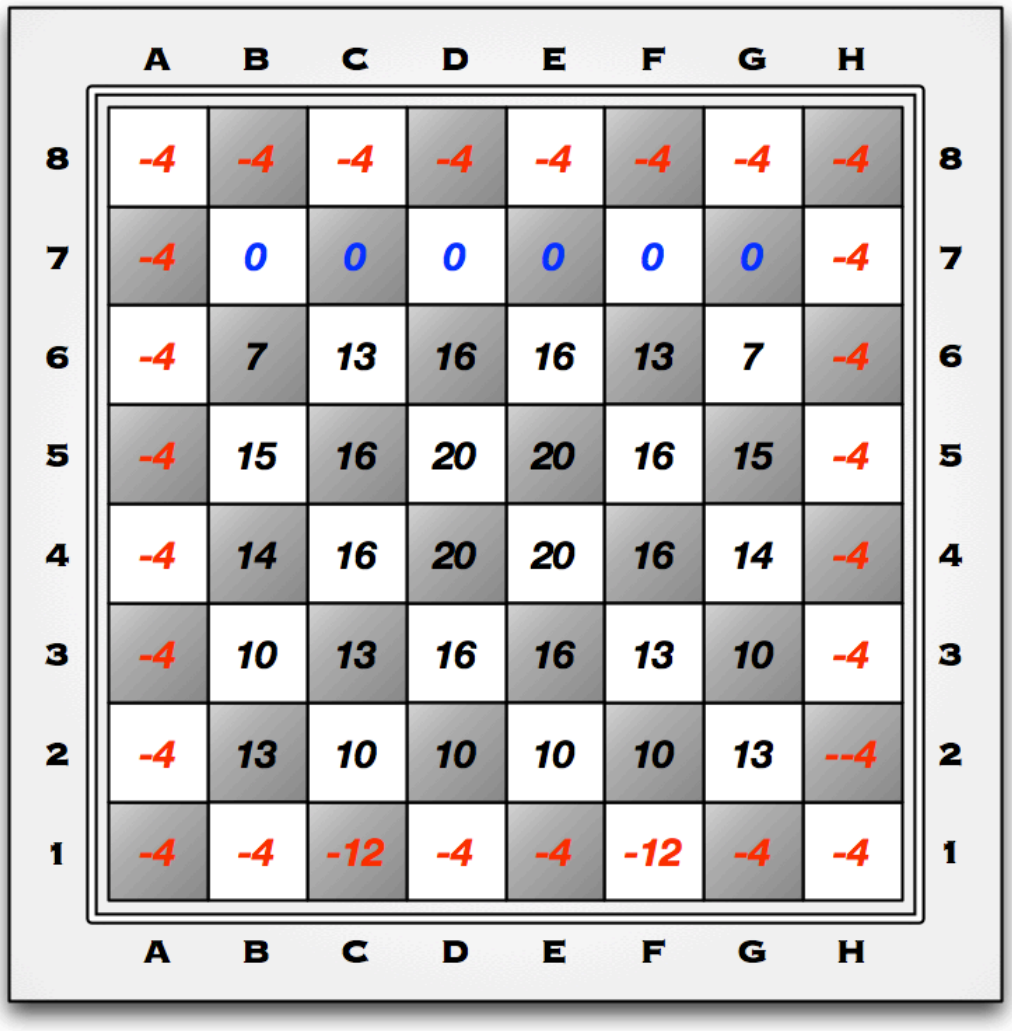
	A	B	C	D	E	F	G	H	
8	0	0	0	0	0	0	0	0	8
7	1	25	30	30	30	30	25	1	7
6	5	25	30	30	30	30	25	5	6
5	5	15	20	20	20	20	15	5	5
4	5	15	10	15	15	10	15	5	4
3	10	10	8	8	8	8	10	10	3
2	-5	-10	-10	-24	-24	-10	-10	-5	2
1	0	0	0	0	0	0	0	0	1
	A	B	C	D	E	F	G	H	

In the end game the central pawns, if they have not done it before, are more encouraged to advance and to occupy the last row in the table. Lateral pawn are more complicated to move and the value is not elevate as the central ones.

BISHOPS

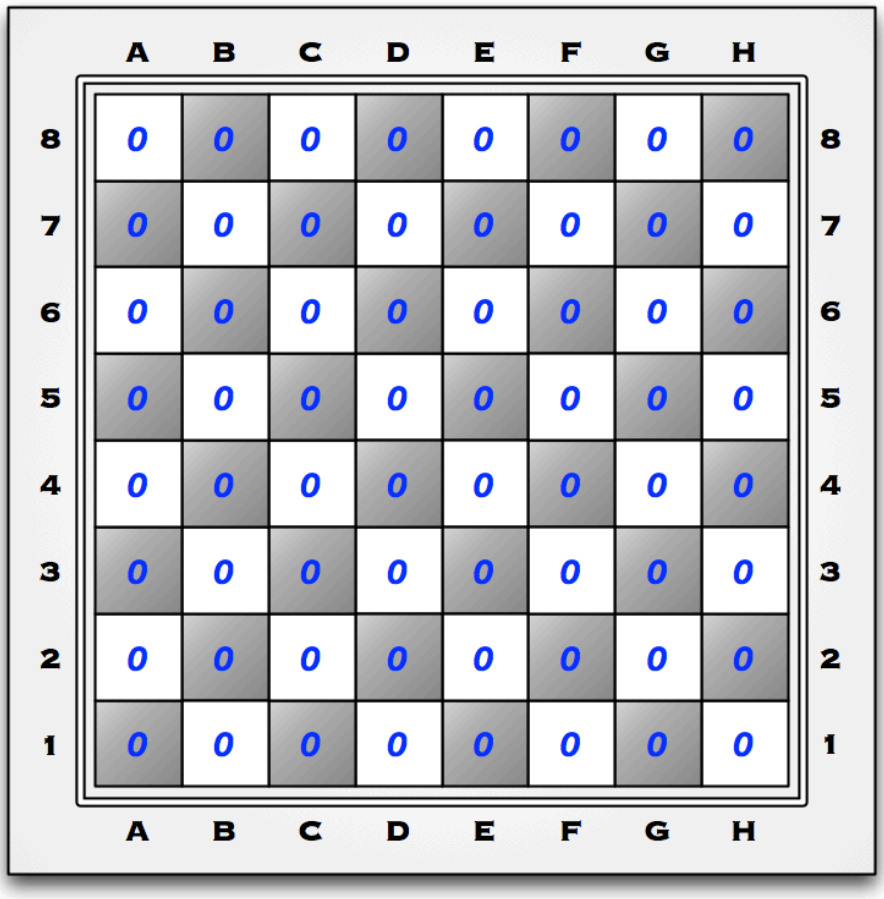
ALL

Bishops has only one table for all the game phase, because their behaviour is much more determinated by other heuristic and value, there's not a best position for the bishops. In general ew encourage the bishops to go in the central zone of the board and to evitate the bound row and column of the chess table.



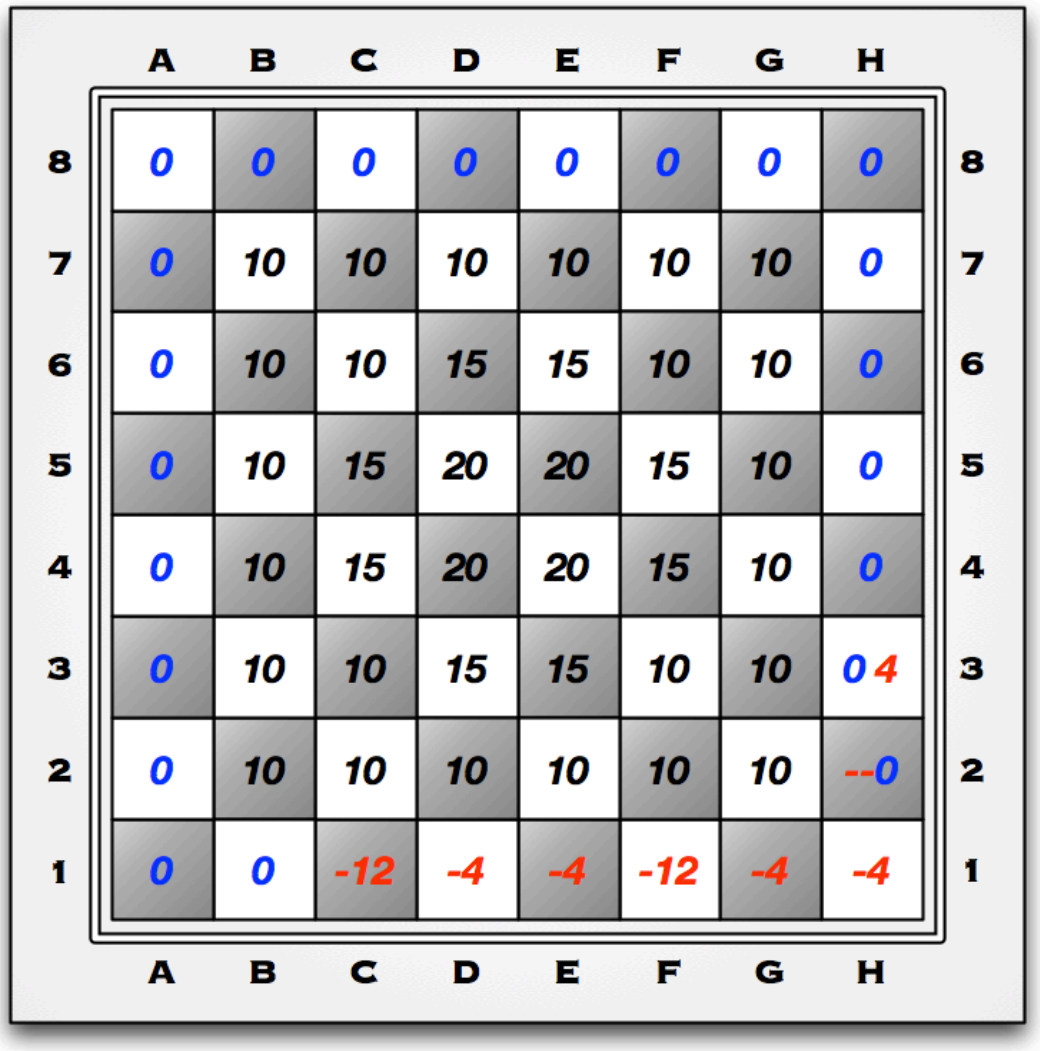
OPENING AND MIDDLE

For the opening and middle game we have not found best position value for the queen that is a complicated piece to move. We left the table to 0, so the heuristic does not count the queen in this game phase.



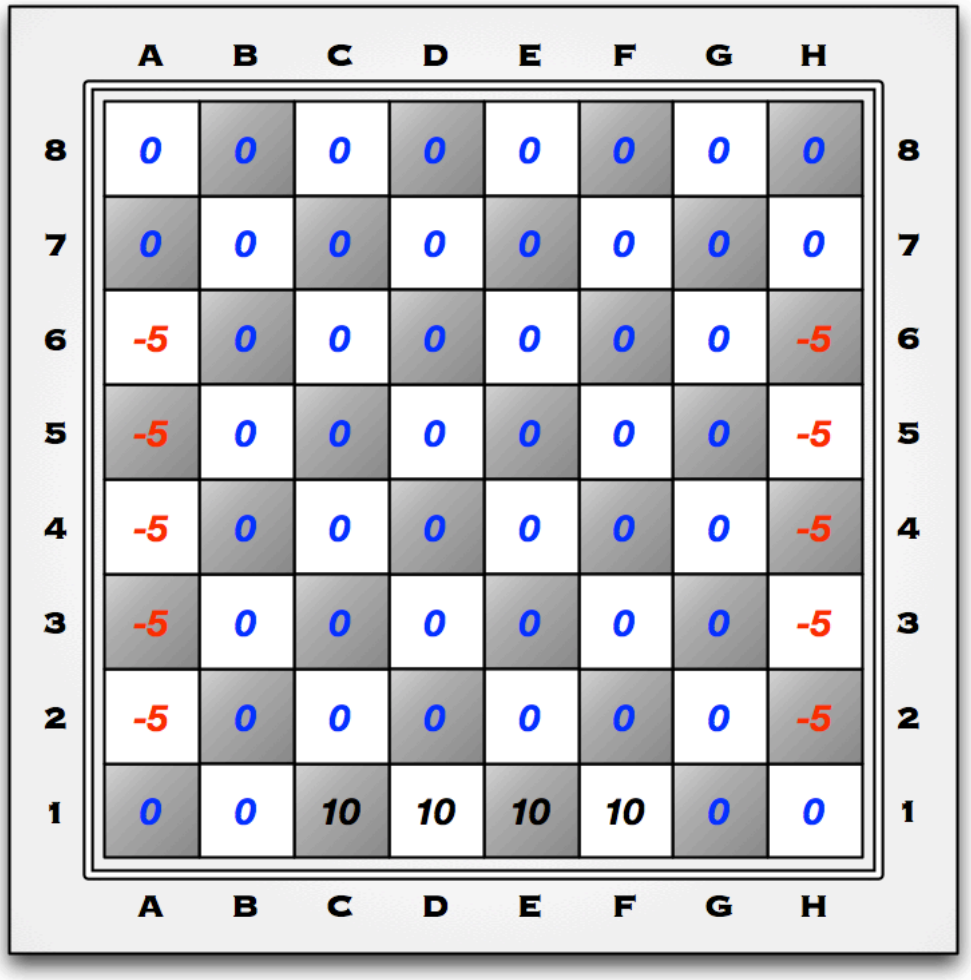
END GAME

The queen in the end game is encouraged to stay mostly in the center, where have the maximum possibilities of movement.



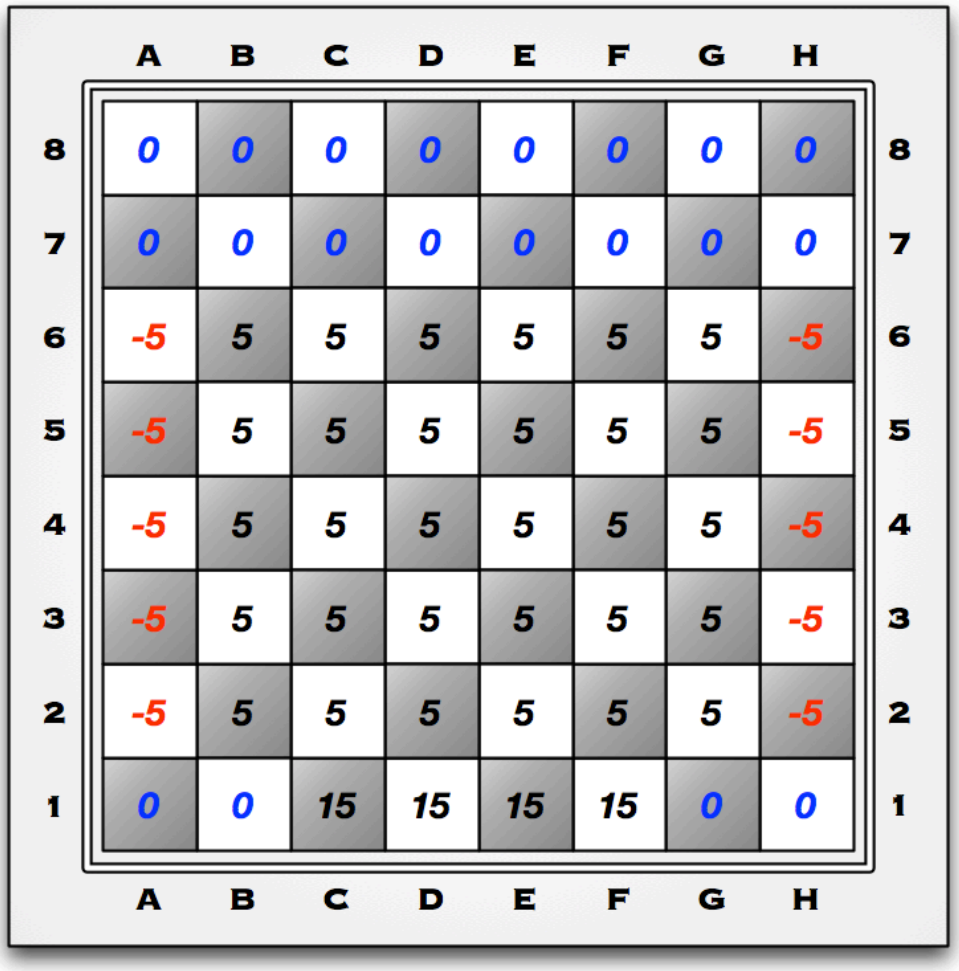
OPENING

The rooks in the opening games are encouraged to not move forward and to stay in the first row.



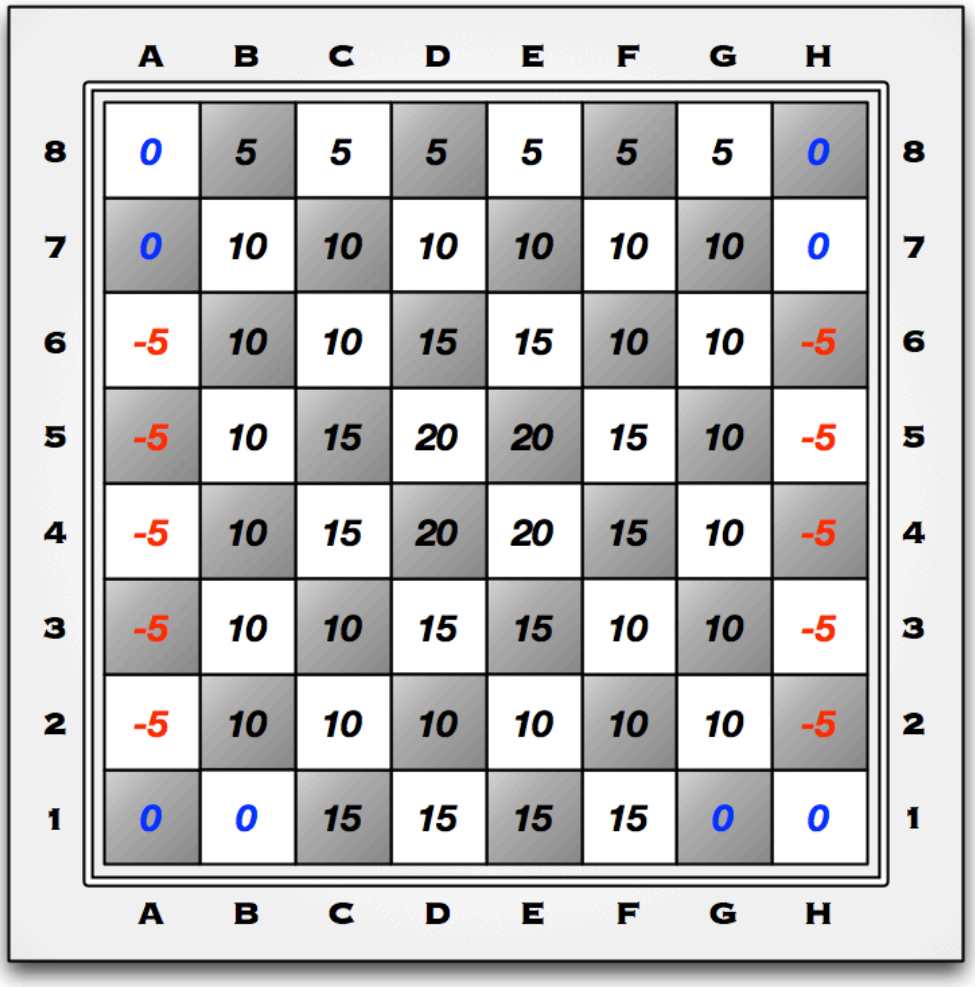
MIDDLE GAME

In the middle the rook are encourage to go high in the center.



END GAME

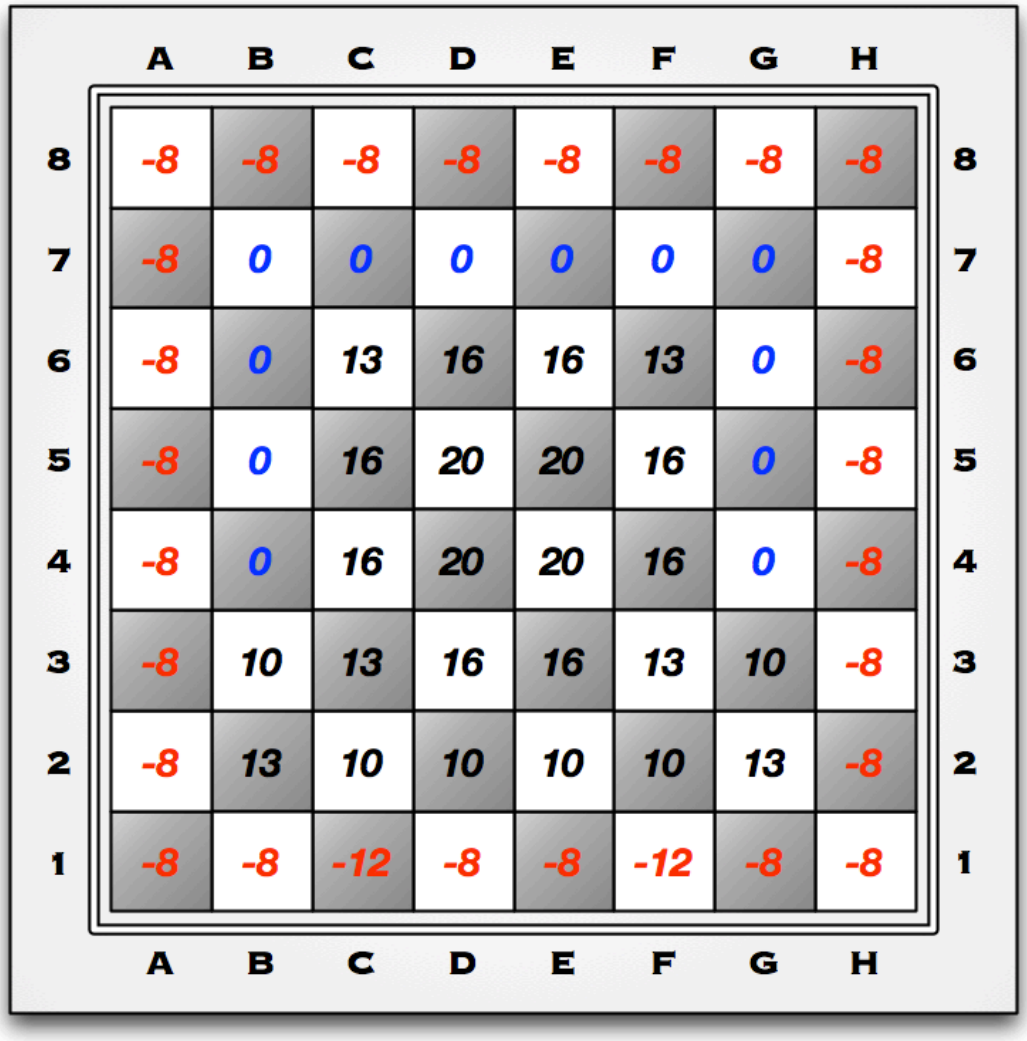
In the end game the rooks are involved to advance, also on the last row, to try to make a check to the enemies king.



	A	B	C	D	E	F	G	H	
8	0	5	5	5	5	5	5	0	8
7	0	10	10	10	10	10	10	0	7
6	-5	10	10	15	15	10	10	-5	6
5	-5	10	15	20	20	15	10	-5	5
4	-5	10	15	20	20	15	10	-5	4
3	-5	10	10	15	15	10	10	-5	3
2	-5	10	10	10	10	10	10	-5	2
1	0	0	15	15	15	15	0	0	1
	A	B	C	D	E	F	G	H	

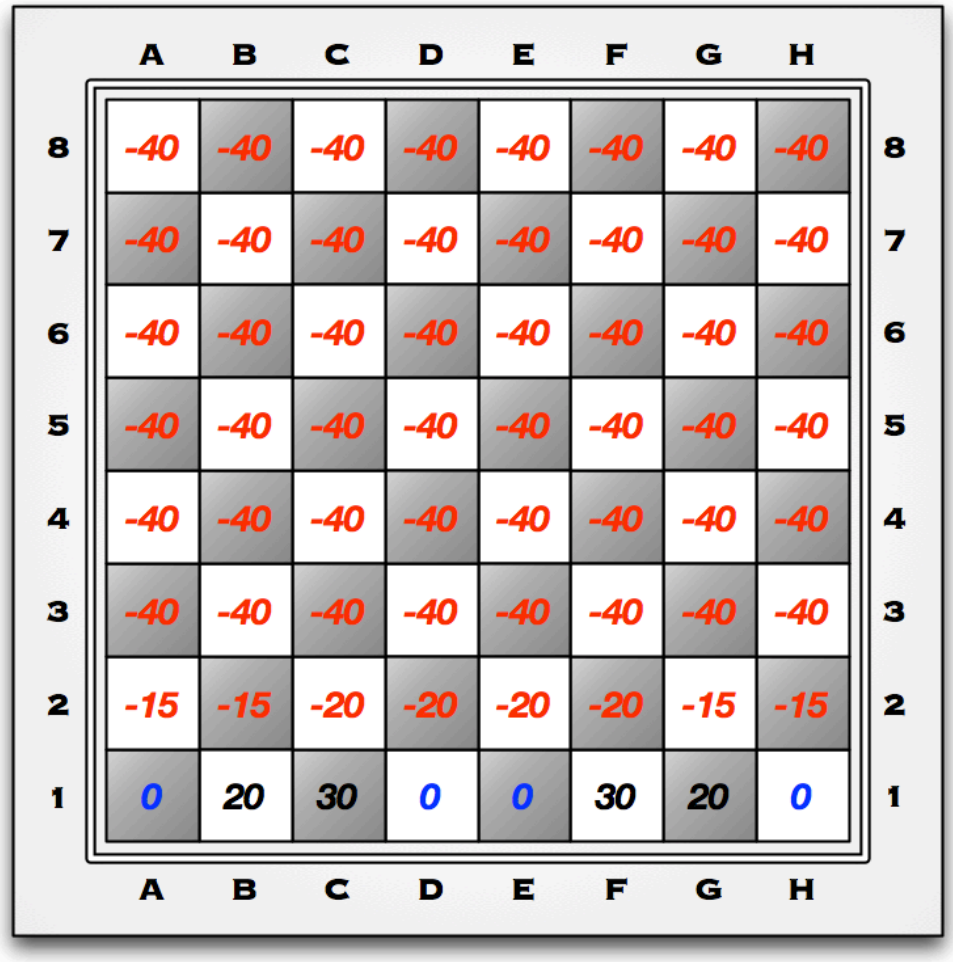
ALL

The knights are pieces, as the queen, for which is very difficult to find a correct value for every game phase. We use one table for all the game phases and encourage the knights to stay in the center of the board and assign a penalty if are not developed.



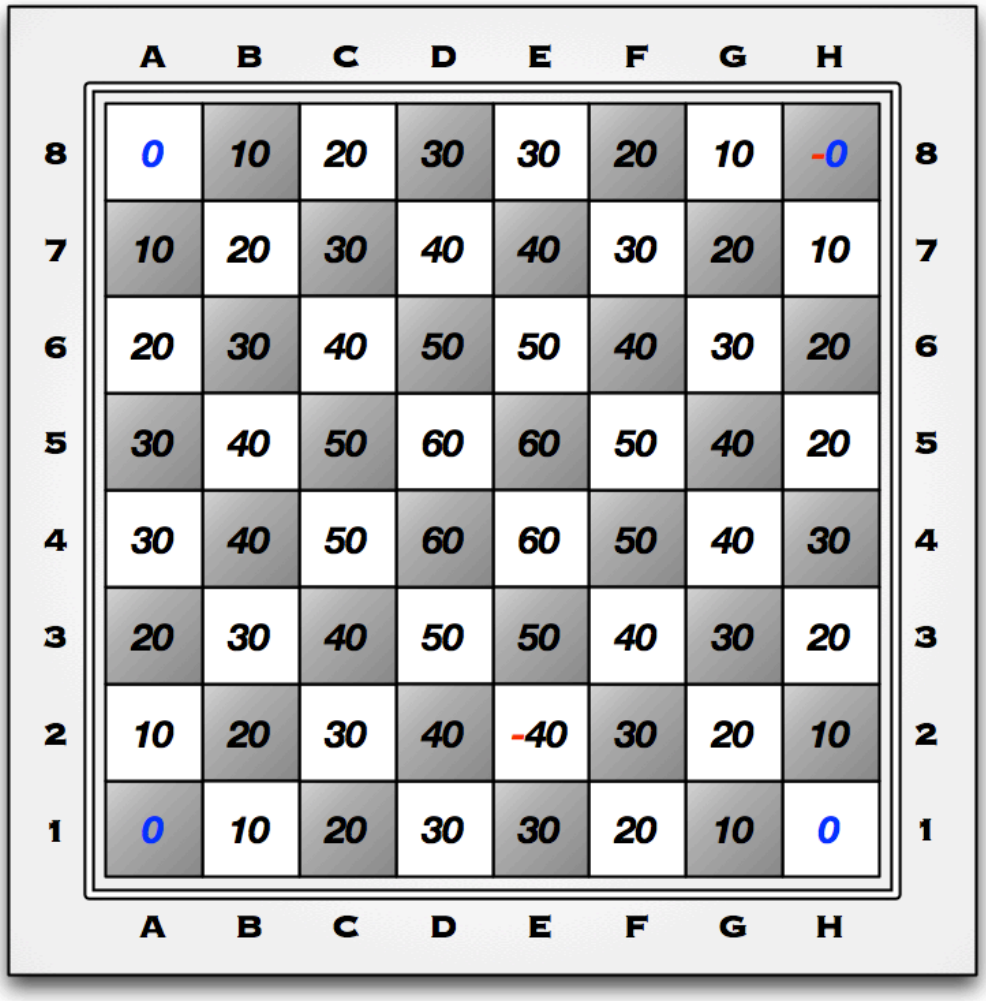
OPENING AND MIDDLE GAME

The king in the opening and middle game has to stay covered in the first row.



END GAME

The end game for the king is a particular situation, it can move in the chess table to be in save from the check. We use this table that encourage the king to stay in the center.



	A	B	C	D	E	F	G	H	
8	0	10	20	30	30	20	10	-0	8
7	10	20	30	40	40	30	20	10	7
6	20	30	40	50	50	40	30	20	6
5	30	40	50	60	60	50	40	20	5
4	30	40	50	60	60	50	40	30	4
3	20	30	40	50	50	40	30	20	3
2	10	20	30	40	-40	30	20	10	2
1	0	10	20	30	30	20	10	0	1
	A	B	C	D	E	F	G	H	

IMPLEMENTATION

We have implemented all the table shown above as array of 64 elements; when we want to evaluate the position of a single piece we calculate where it's positioned, and use this value (is a value that go from 0 to 63) as an index of the array of these piece in the current game phase. Since all these table are from white point of view, if we want to calculate the value of positon for the black pieces, revert its position on the board in white point of view by doing 64-*calculated_position*.
Once we have calculate the value of position heuristic for both the white and black pieces, we calculate the final value that is:

We are **white: Position= Position_White - Position_Black**.

We are **black: Position= Position_Black - Position_White.**

It is, as before, a negative value if the position of my adversary is better than mine, and positive otherwise.

MENACED HEURISTIC

This heuristic evaluate a particular game situation by calculating the number and the type of piece that we can menace.

The calculation of the number and type of pieces menaced required an extra computational effort: for every piece type we generate the following movement, as mean generate another level in the search tree. Once generated all the movement, we check if in this movement there are any piece killed, and if it is we save the type of piece.

The killing of a piece in this level of the tree means that this piece was menaced in the level before, the one that we are examining. So we look at how many piece we can menace with every movement and we calculate the value of this heuristic by multiply the number of menaced pieces of every type for the weight that we assign to every type menaced, as reported in the next line.

King: 300

Queen: 45

Rook: 25

Bishop: 17

Knight: 15

Pawn: 5

Once calculate the value of menaced for black and white, we calculate the final value of the heuristic:

We are **white: Menaced= Menaced_White - Menaced_Black**.

We are **black: Menaced= Menaced_Black - Menaced_White.**

CHECK HEURISTIC

This is the heuristic function that individuate if we lost the king in the last level of the search tree (in the other level there's the function *ourCheck()* that make this control). This simply look if the bitboard that represent our king is equal to 0; if this happen, we have lost the king and we return an high negative value.

This function is used to evaluate if it is necessary to apply all the heuristics function or if we can skip it and return an hig negative value, corresponding to the lost of the king.

INTEGRATION OF DIFFERENT HEURISTIC IN A UNIQUE FUNCTION

After the calculation of all the different heuristic, we integrate all of them in an unique function, that is the following:

$$Heuristic=(a1\cdot material + a2\cdot position + a3\cdot menaced)\cdot sideToMove(d)$$

The function side to move is used to give the correct sign to the heuristic value, according to the depth of the search tree.

The value material, position and menaced are the heuristic calculated before; because these are different type of heuristic and every of these has pros and cons, we have added 3 multiplicative constant to assign different weight to every heuristic.

The material is an important heuristic that is based on what piece we kill on a path and also give information on our health situation. If we have a lower material value than our enemies, maybe this is not a good leaf node.

The position heuristic is useful at the beggining of the game, to make all the pieces like the pawns to advance, but it is unuseful in a complex situation like the endgame.
The menaced heuristic is almost useful in all the game situation because can permit us to "look forward" and individuate if a movement can menace a high number of pieces and make the choose of the right movement complex for our enemy.

For all of these factor the mutliplicative constant *a1*, *a2* and *a3* are changed during the game, according to the game phases: in the initial and middle game phase we put our emphasis on the position and menaced pieces, to make the pieces advance and kill.
In the last phase of the game we enlarge the weight of the material heuristic, because it's more important to do not lose piece of high weight in the board, like the queen or the rook, that are very important in this phase of the game.

The implementation of the algorithm is the one that can be seen below:

```
int heuristic (int depth, Board * board)
1.  {
2.          int phase= calculate_game_phase(depth);
3.          if(ourCheck(board) != KING_IN:CHECK)
4.          {
5.                  int material = material_heuristic(board);
6.                  int menaced  = menaced_heuristic(board);
7.                  int position = position_heuristic(board);
8.
9.                  if(phase == END)
10.                         return (a1_end * material + a2_end
    * position
11.                                 + a3_end * menaced) *
    sideToMove(depth);
12.
13.                 else
14.                         return (a1 * material + a2 *
    position
15.                                 + a3 * menaced) *
    sideToMove(depth);
16.         }
17.         else
18.         return KING_IN_CHECK * sideToMove(depth);
19. }
```

In the line 4 we control if we have lost the king, and in this case we do not evaluate other heuristic, but we return immediately the constant KING_IN_CHECK that has a high negative value (line 18).

In the other case we calculate (line 5, 6, 7) the value of the heuristics and multiply for the constant multiplicative *a1*, *a2* and *a3* that vary, depending from the game phase.

EVALUATION OF THE SOFTWARE

In this section we will review few benchmarking tests that we have done.

TIME EVALUATION

Before we implemented the prune algorithm and the heuristics we made a benchmarking test with two different computers.

	Depth	Time
Macbook Pro	5	10 seconds
Macbook Pro	8	1 minute
Macbook Pro	9	13 minutes
AlienWare M11x	5	1 second
AlienWare M11x	8	10 seconds
AlienWare M11x	9	4 minutes and 30 seconds

The specifications of the two computers are:

- **AlienWare M11x:**

  Intel Core i7 CPU 1.20GHz, RAM 4GB DDR3, x64

- **Mackbook Pro**

  Intel Core 2 Duo 2.4 GHz, RAM 4GB DDR3, x64

The truth is that we are really proud of this results and we think that this could be achieved thanks to all the performance optimizations that we have done all along the practice.

As we have already explained in the first sections, the state space of one chess game is given by this expression: b^m where b is the number of branches and m is the depth of the search tree.

Even if we only go to a fifth depth level, this number of nodes would be unaffordable. For this reason we only maintain in memory the total number of each node, so we only have 137 nodes in memory at every chess movement.

As we've already explained the size of a node is about 512 bytes, 0.5 kB. So the total amount of required memory space without any prune is of $137\ nodes \cdot 0.5kB \approx 70kB$.

To test the software implementation, we create functions that are dedicated to test the NegaMax search and the move generation process.

In this way, the file ‘move_test.c’ and ‘move_test.h’, are instructed to test the movements of each piece, starting from a Board passed as parameter.

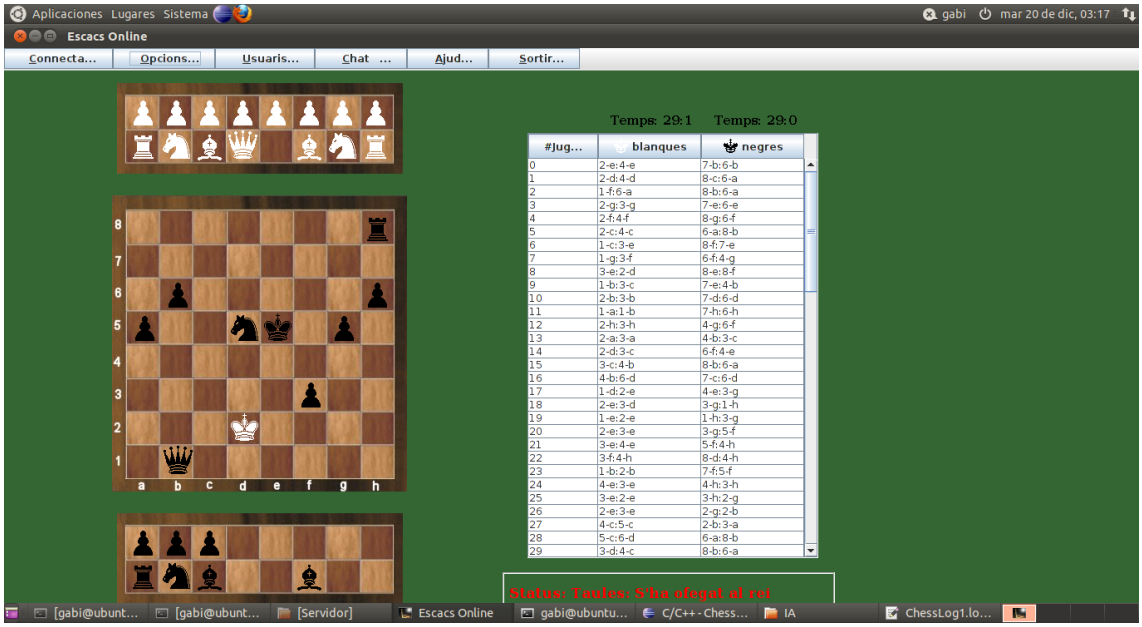
The file ‘search_test.c’ and ‘search_test.h’ are instructed to test the search: starting from a Board, we generate with the heuristic the best next movement. In this file we also test that the search algorithm can also work with simple and different heuristic.

One more test that we done was to follow a chess match, using a match log. This log shows the movements of each player and all the modification that are produced on the board.

Now, we show two example of match log.

The first log “ChessLog1.log” shows a checkmate of the white pieces in 16 movements.

The second “ChessLog2.log” shows a table with the drowing of the white king.



Following with the tests, we must recall that we programmed a complete log library that allowed us trace all the piece movements with great ease. Given that we work at bit level, it whould have been hard to trace all pieces, but thanks a logBoard() function that we programmed, we could print a really intuitive representation of the board into the selected file.

So for printing a board state we achieved something like this:

```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        1 1 1 1 1 1 1 1
        1 1 1 1 1 1 1 1
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 1 0 0 0
        1 1 1 1 0 1 1 1
        1 1 1 1 1 1 1 1
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

This log library also allowed us to write logs with a different log level priority, and also allowed us to deactivate all log operations just activating one constant. This doesn’t mean that we stop writing in the file, this means that the log library functions are not even processed, so there is no waste of processor time.

CONCLUSIONS

As we think about it, we realize that this practice is one of the most similar to what would be a real project.

As we develop the chess player we used a lot of practices used in agile methodologies. We have used a source repository to have the same code for all the group members. This behavior let us synchronize very well among us. But although we have used a code repository, we have been working in group as in the pair programming agile methodology practice.

Furthermore, this is the first practice that implies to write a very good code in order to take advantatge of the performance tricks that one can insert in his code. We have focused all our chess computer player in perform his moves as fast as possible. For this reason we have worked with bitmaps and we have implemented and tested algorithms such as Negascout and Negamax.

Once one looks to the internals of many advanced software, he will find a lot of bit level operations such as ands, ors, xors, etc. Before this practice, one could be afraid of all this low level trickery, but right now we are able to fully understand all that kind of operations. At first it was difficult to realize what operations we were doing with our bitmaps, but now we think that it is easier and faster for the programmer to work with bitmaps than with matrix. We have improved as engineers.

To finalize with the optimization concept, we are really proud to have achieved the search in the tree to go until the sixth level in a very appropriate time for a chess game. As we have seen, the average level that an amateur chess programmer can go down in the search is until a fourth level, so this means that all our focus in the efficiency has paid off.

Another interesting fact is that we were able to adapt a very complex recursive algorithm that we found in many academic papers… And it worked! We were able to adapt concepts thought by other people and adapt them to our particular case and implementation. We have realized that this is the way that scientific community should behave. And this is the way that science advances.

Talking strictly about our experience, we have found the practice to be very time consuming. We were really exhausted at the time we had finished it. We had worked hard through many weeks. We estimate that we had worked about 113 hours, what comes from multiplying two weeks with 8 hours per week. We know that we have not worked eight hours every day of these two weeks, but the truth is that we have worked more than two weeks.

The thing is that we think that could be a good option to delay the deadline to the exam dates in order to achieve a better result in our chess computer player. Feels sorry to could not achieve better results because of time. We wanted to do more optimizations to the code, such as controlling the remaining time for the game to end and proportionally increase the depth level that our algorithm would search. We also were not able to improve our checkmate strategies because of the lack of time.

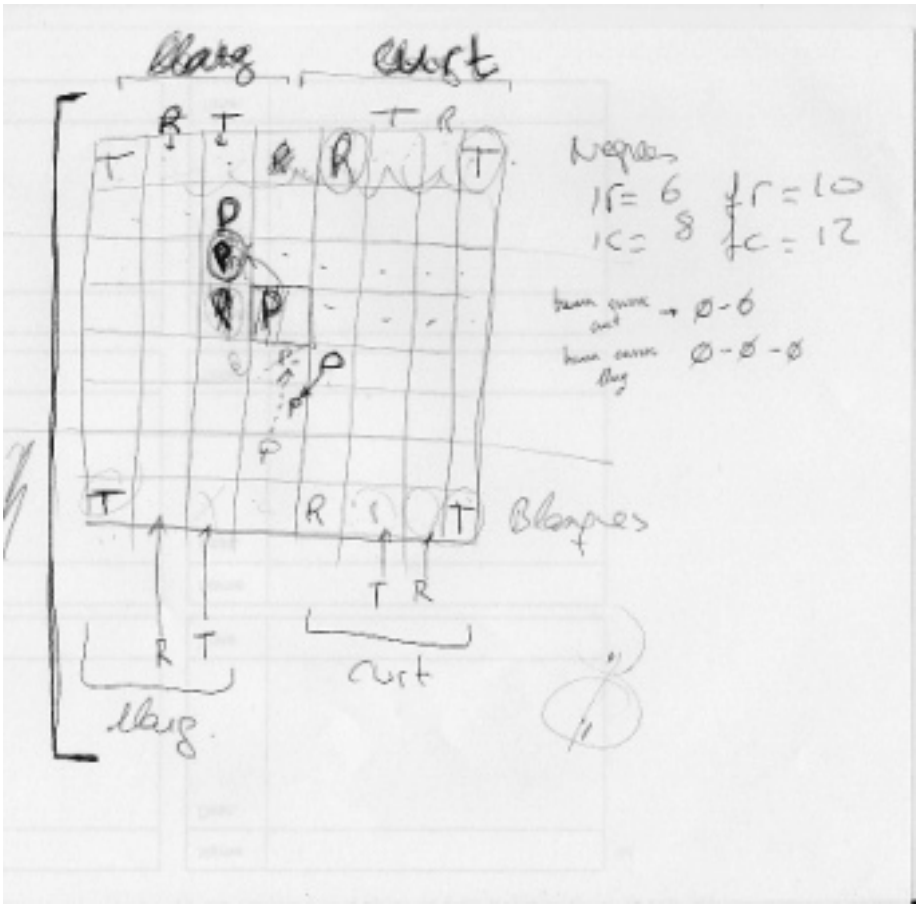
Finally, a curious fact about our heuristics. We think that our heuristics are not really bad because surprisingly our computer chess player began the game with the shepherd checkmate

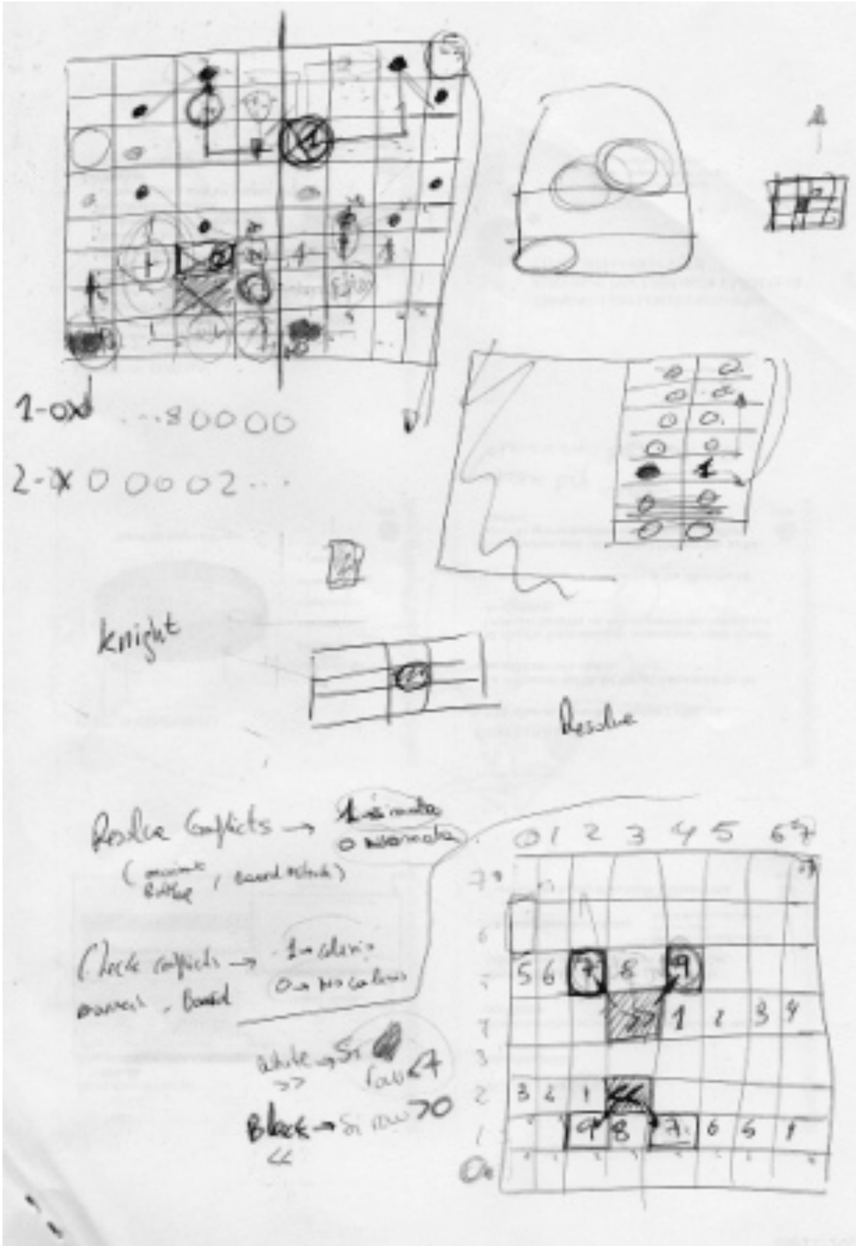
without being particularly programmed for it. It was a logical movement that the computer found due to our good heuristic strategies.

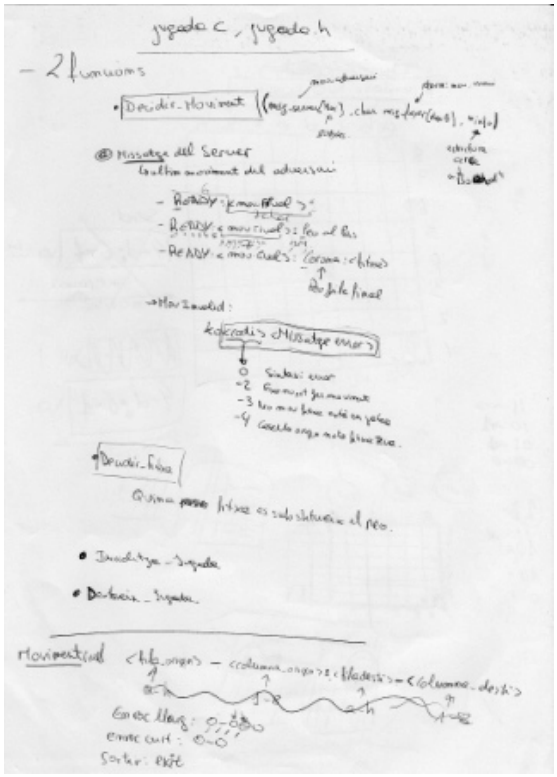
To sum up, we think that this practice is really interesting but it would be perfect to have many more days to end it.

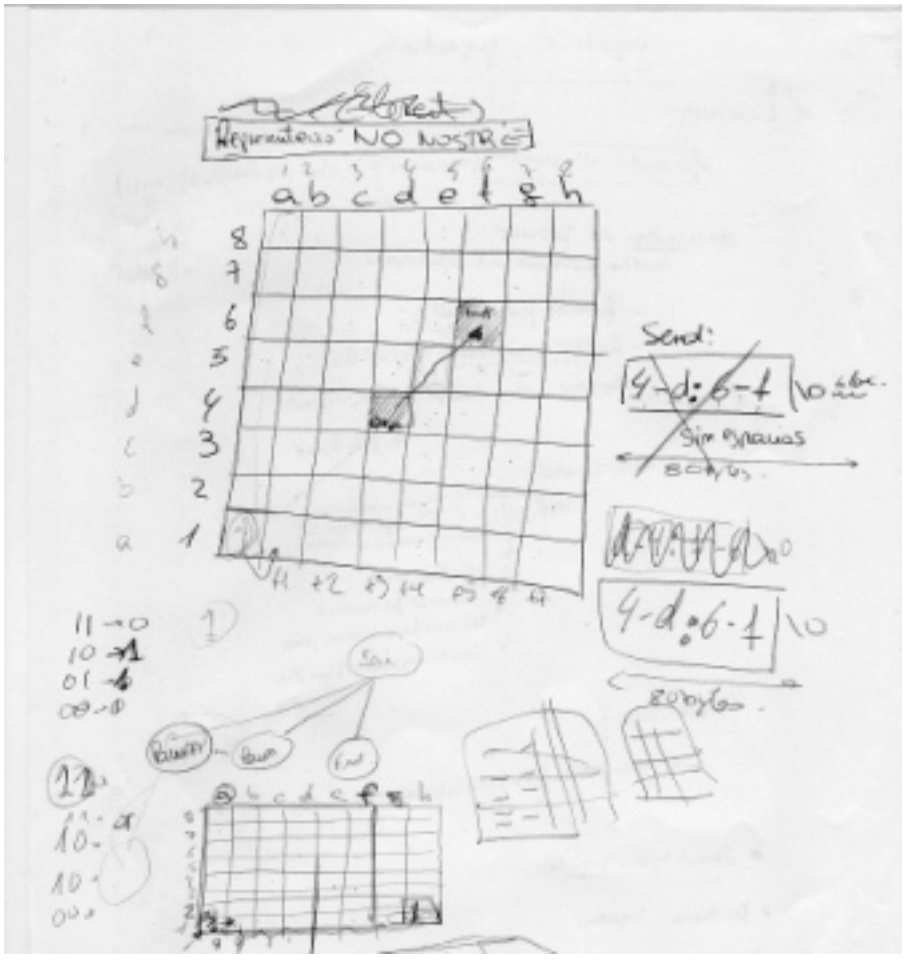
ANNEX I

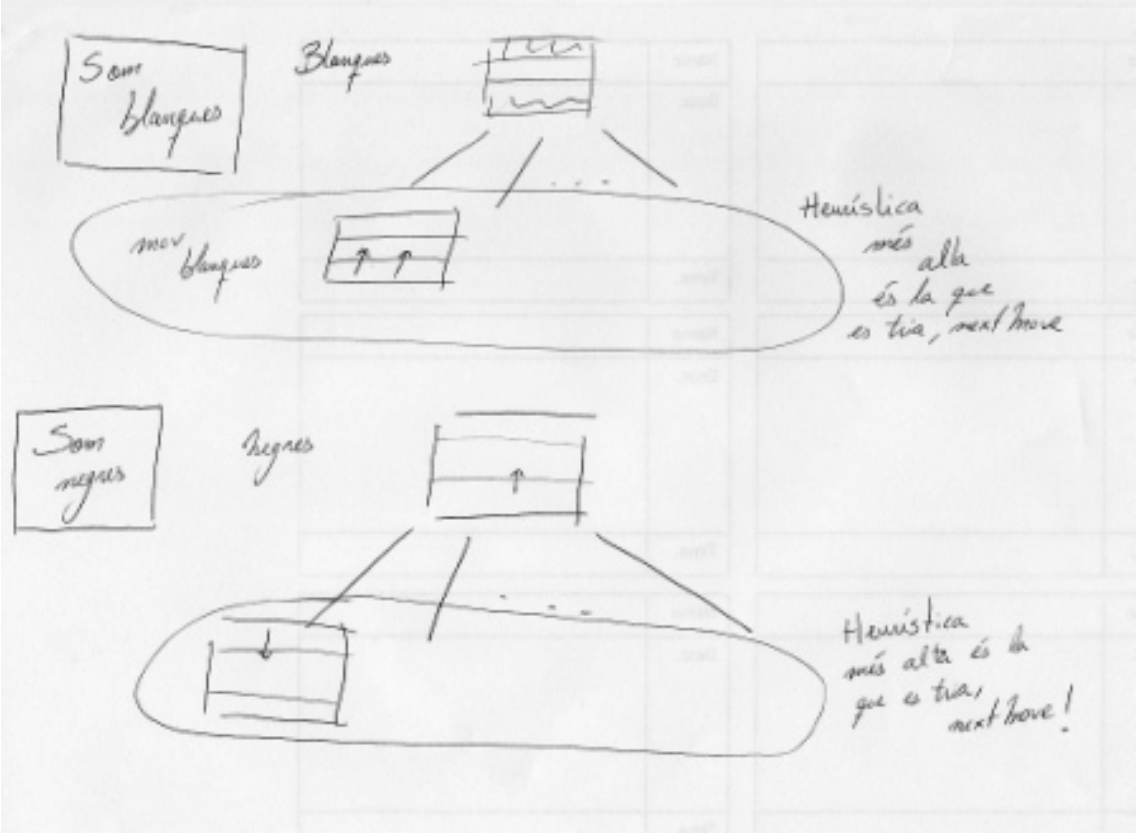
As a curiosity, this annex will contain the major part of the drafts we have used in our discussions.

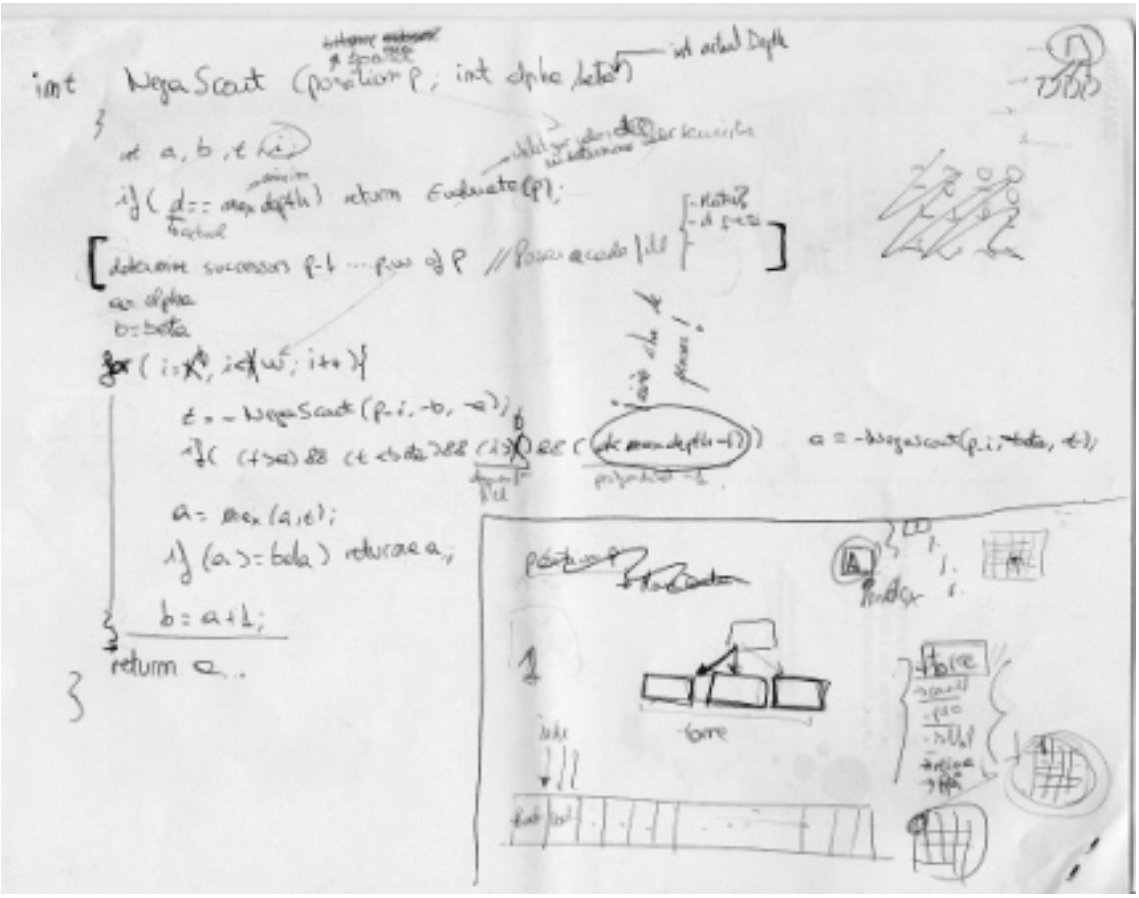


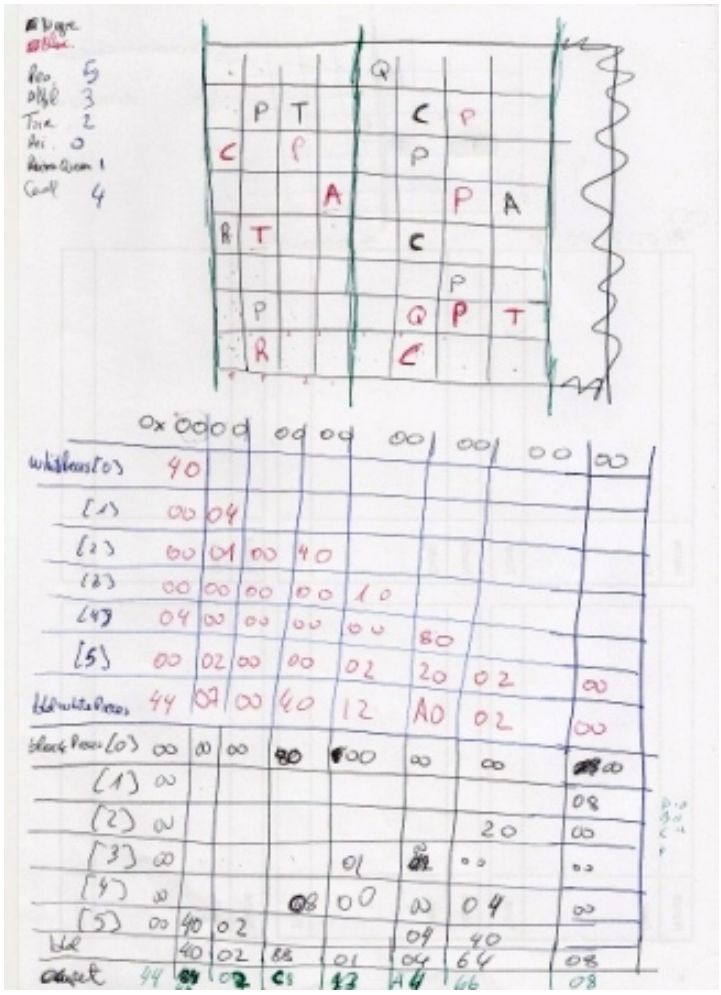


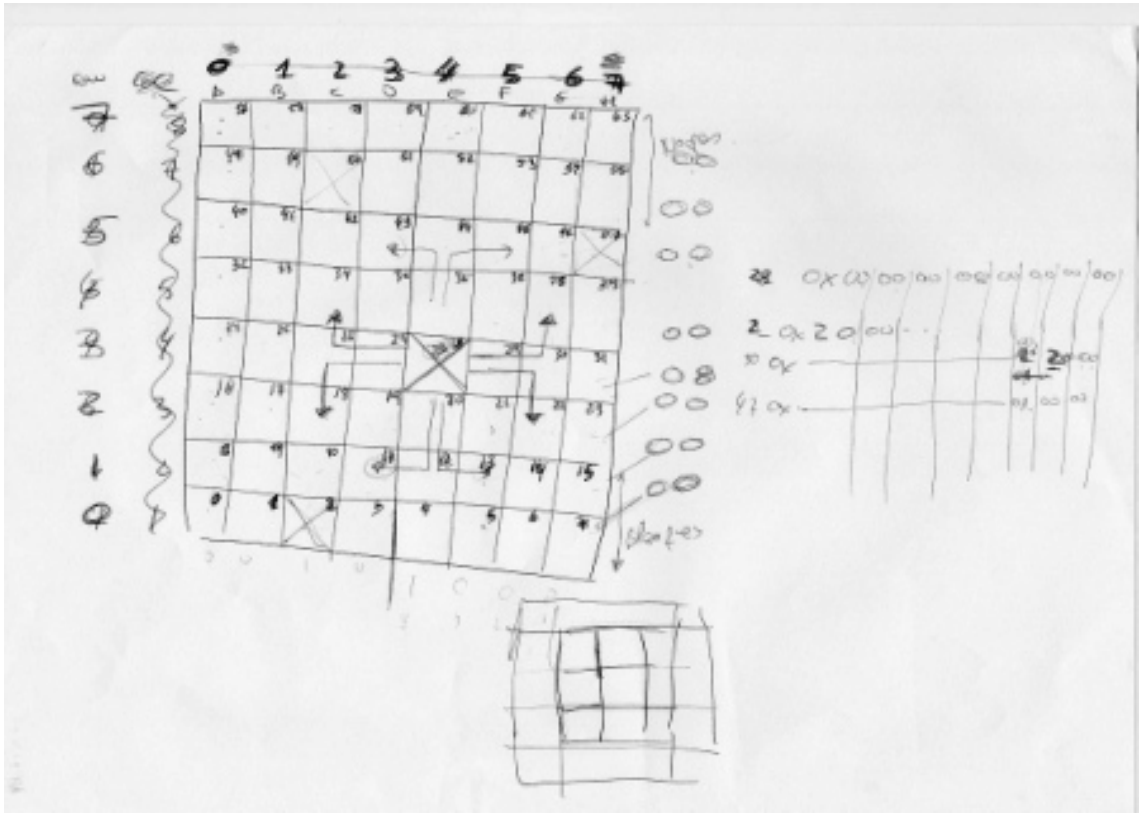

jugada c, jugada h
- 2 funcions
Decidir_Moviment
Missatge del Server
Decidir_Fitxa
Moviment:

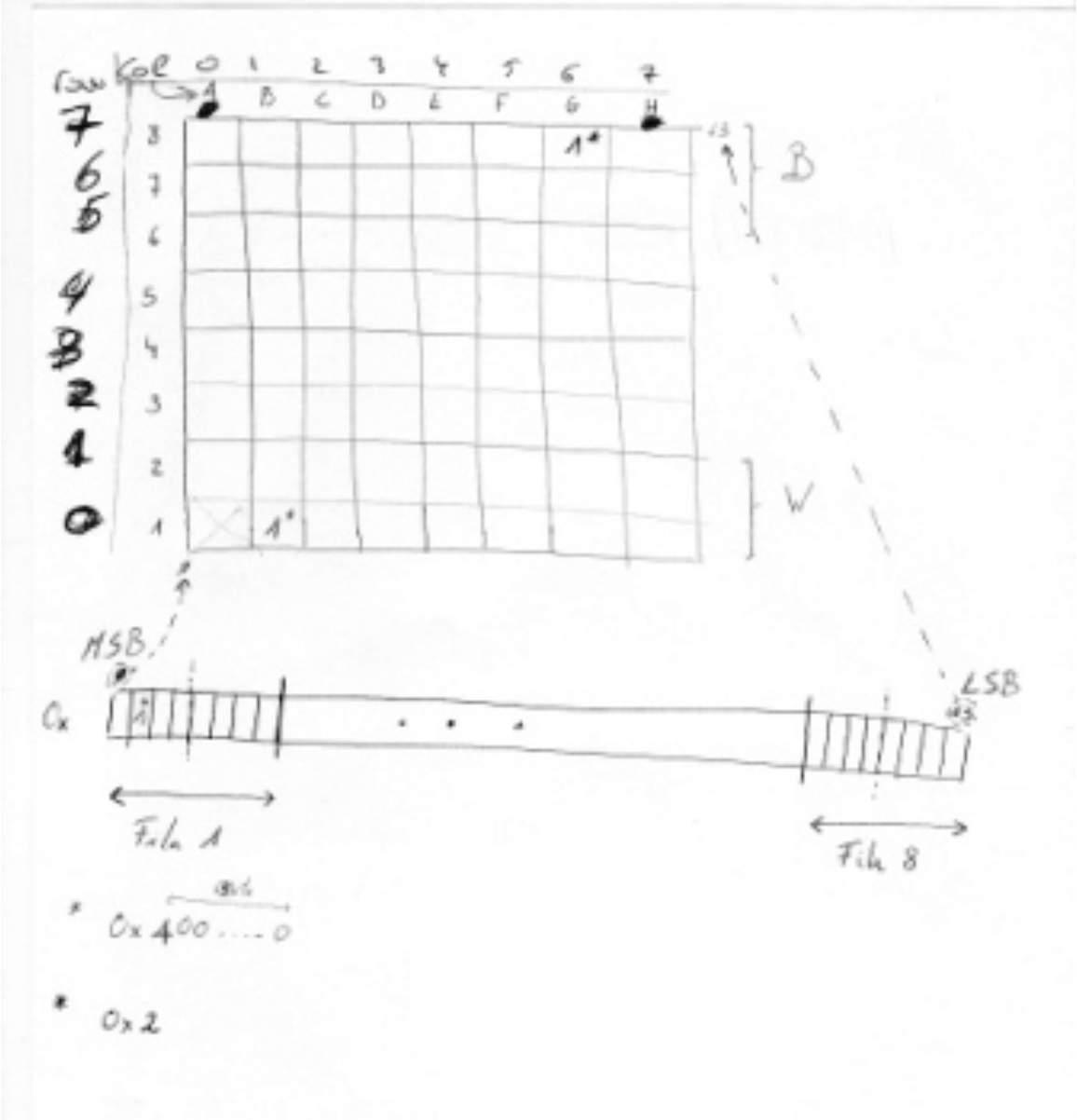

Representació NO NOSTRE
a b c d e f g h
8 7 6 5 4 3 2 1
Send:


Son Blanques
Blanques
mov Blanques
Heurística més alta és la que es tria, next move
Son negres
Negres
Heurística més alta és la que es tria, next move!


int NegaScout (position P; int alpha, beta)
int actual Depth
int a, b, t, i;
if (d == max depth) return Evaluate(P);
actual
[determine successors P-1 ... P-w of P // Possible moves]
a= alpha
b= beta
for (i=1; i<=w; i++){
t = - NegaScout (P-i, -b, -a);
if ((t>a) && (t<beta) && (i>1) && (d < max depth -1))
a = -NegaScout(P-i, -beta, -t);
a = max (a,t);
if (a >= beta) return a;
b = a+1;
return a;
}
1
Torre






Row
Col
0 1 2 3 4 5 6 7
A B C D E F G H
7 6 5 4 3 2 1 0
B
W
MSB
LSB
Cx
Fila 1
Fila 8
Cx 400....0
Cx 2