A discussion on the making of HIPP

By Marcel Vlastuin

I have created HIPP[†] to play the game of Caïssa for my participation in the Codecup 2003. The final contest of the Codecup on January 15th was won by HIPP. I will describe how HIPP works and will explain its most important features to you. I have written HIPP in C. I will try to limit the number of examples of source code to explain what I have done.

I will do this in the following order:

- How to make the game Caïssa?
- The basic idea of how to make HIPP good
- The implementation of MTD(f) and alfabeta pruning
- How to analyze as deeply as possible?
  1. Don't analyze at all
  2. Don't check the connection rule at all
  3. Go deeper when there are fewer moves available
  4. Go deeper when check is given
  5. Make it faster by going deeper!
  6. Make the search window as small as possible
- How to make an adequate evaluation function?
  1. How to restrict the returned evaluation scores?
  2. Not to stop in a check situation!
- How to store and play moves?
  1. The storage of the pieces and the tiles
  2. Stack and stack pointer explained
  3. The do & undo function
- How to generate moves?
  1. The main structure of the generator of moves
  2. Using pointers in look-up tables
  3. Using pointers to pointers in look-up tables
  4. Generating the queen moves
  5. Generating the rook and bishop moves in check situations
  6. Generating the rook and bishop moves in non-check situations
  7. Generating the knight moves
  8. Dealing with swap moves of the rook, bishop or knight
  9. My solution for the connection rule
- Using an opening book

Enjoy and learn,

Marcel.

[†] Actually HIPP stands for DELL. HIPP supposes to be the son of HAL, which is in Stanley Kubrick's *A Space Odyssey 2001* an arrogant IBM series 9000 computer.
You can download a fully documented version of HIPP on my website www. vlastuin.net.

How to make the game Caïssa?

Caïssa is a game of strategy and was introduced by Christian Freeling[‡]. Because it is a game of strategy, I have chosen to apply the minimax algorithm. This paper is about the implementation of this algorithm. Brute force is used to find a continuation that leads to a definite advantage. In the evaluation function you can apply your ideas of strategy to achieve a better position than your opponent to develop the game in your favor.

Everything described in this paper is a sort of grab bag containing my ideas. Most of these ideas are best qualified as: 'rather a bad plan than no plan.' You might find some of them useful.

The basic idea of how to make HIPP good

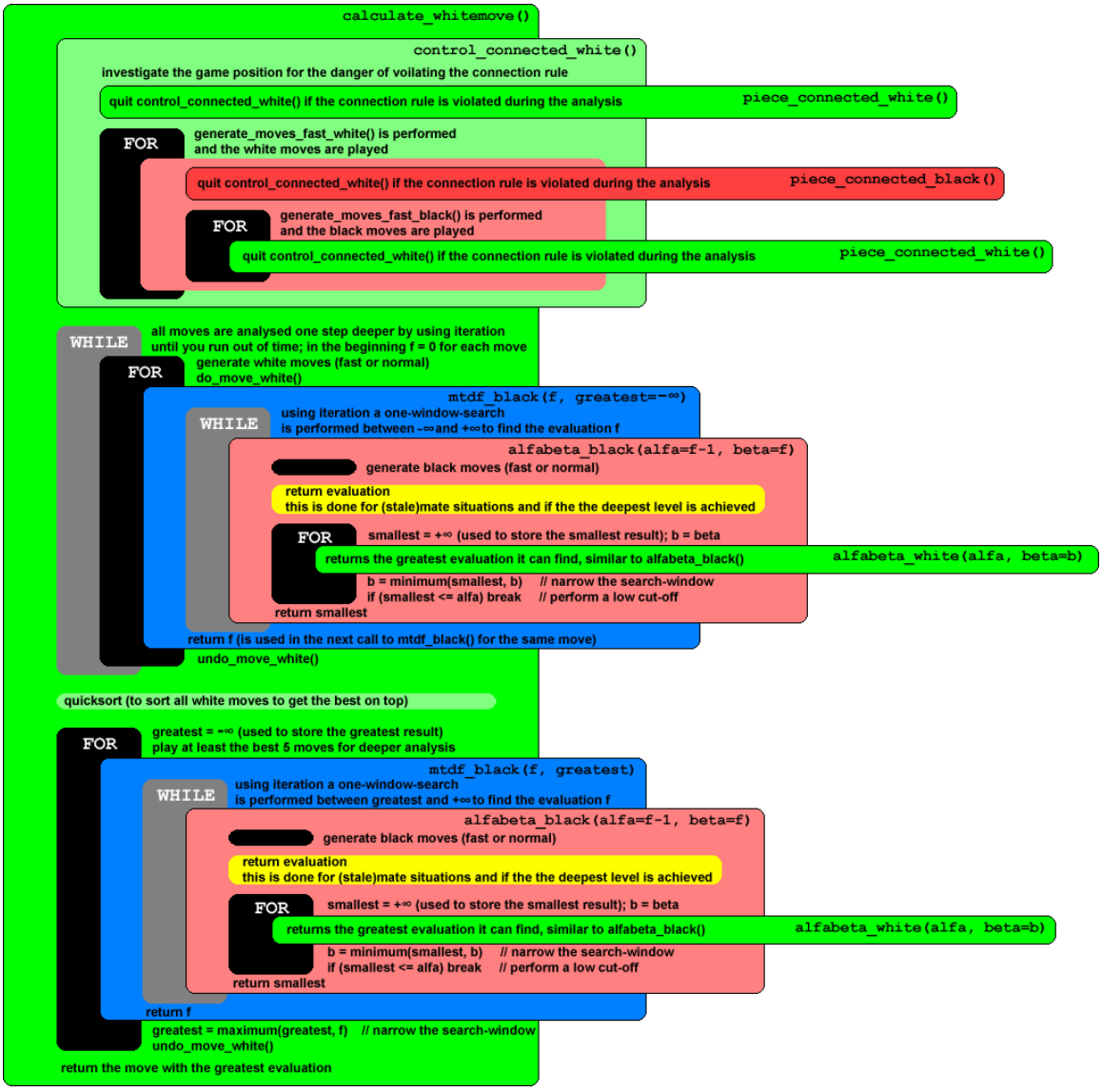
In general I believe that brute force is the best for games of strategy. So I have optimized the minimax algorithm for speed and efficiency. Most of the time spent on HIPP was put in designing and programming the generator for moves. I used 'smart' look-up tables to get the speed I wanted. HIPP does find legal moves and stores it in roughly 200 CPU clock ticks per move. I think that is pretty good. To improve the efficiency of the algorithm I have implemented the MTD(f) algorithm[§] enhanced with alfabeta pruning but without the memory function. A brief introduction is found at http://www.cs.vu.nl/~aske/mtdf.html. It is a very essential document which you should read first.

I don't believe too much in sophisticated evaluation functions. It is very important though, to use one! I prefer a simple and above all a fast evaluation. If you want to win, you need to maximize (or minimize) something. If playing Othello and Checkers you want more pieces of your own on the board than your opponent. For Caïssa I have applied the same principle. If you cannot capture the queen of your opponent you want at least to capture another piece. And if you are not able to manage that you want to have more moves available for your own pieces than for your opponent. So there is always something to fight for… I have made an evaluation that achieves progression in that particular order; besides a small penalty for putting the queen in the corner there is nothing more in it. This basic idea works well.

[‡] Chistian Freeling and Ed van Zon are the webmasters of www.mindsports.net. On the site you can play several abstract games. Caïssa was introduced by Christian Freeling on this site in 1997.
[§] Aske Plaat: Research Re: search & Re-search, *PhD Thesis*, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, June 20, 1996.

The implementation of MTD(f) and alfabeta pruning

In the scheme below the structure of the routine **calculate_whitemove()** has been visualized. To fully understand it you need to know that from here the recursive routine **alfabeta_black()** is called; this routine is calling **alfabeta_white()** and so forth (this is not shown in the scheme). The recursion is repeated until the desired depth is achieved. Everything in the scheme that is done from the perspective of the white player is colored green and for the black player red. The routine **mtdf_black()** which is called by **calculate_whitemove()** is colored blue. This small routine makes several calls to **alfabeta_black(using: alfa=f-1, beta=f)** to search for the desired evaluation. A zero-window search is used in the last call: it only fails low or high. If a full-window search like **alfabeta_black(using: alfa=-∞, beta=+∞)** were be called directly by **calculate_whitemove()** the same evaluation would be obtained; it would only take more time to get it.



Before the analysis starts, I will investigate if the connection rule forbids certain moves. The idea behind this is to generate moves without checking for the violation of the connection rule. Therefore I have derived a fast version of the move generator. That makes the analysis in the beginning of the game about 10 times faster. If one violation is observed at this stage, the regular generator will be used because of that occurrence. The first violation happens on average after 30 moves (from both players) from the start. For this investigation the game is analyzed two moves deep.

Firstly each move is analyzed one step deep with an initial evaluation f = 0. The MTD(f) routine returns a more accurate value for f. In the second run each move is analyzed two steps deep with the stored evaluation f. This procedure continues as long there is analysis time left. In every analysis a full-window search is performed by MTD(f).

Secondly the best five moves (and if there is time left even more) are analyzed one step deeper. Therefore the moves have been sorted first by evaluation value. The best move is analyzed first using its previous evaluation f. With the new result a smaller window search is performed for the next moves. Because the move that was probably best was analyzed first the next moves will soon fail low in its analysis. This effect, called *killer moves* reduces the analysis time dramatically.

How to analyze as deeply as possible?

Besides using regular techniques such as alfabeta pruning and MTD(f) a lot of small techniques can be used to get deeper, within the same amount of time.

Don't analyze at all
If only one move is available just play it. The time spent on it could be used well if two or more moves are available the next time. So this decision leads to deeper analysis.
If initially it becomes clear that some moves lead to a loss don't analyze them anymore. In that case the number of moves to be analyzed decreases. If that number decreases to one the only move left is played immediately.

Don't check the connection rule at all
As described in the last paragraph the biggest advantage can be achieved by not checking for the violation of the connection rule. The advantage is an analysis which is a fully step deeper.
There is hardly any risk at all using this technique because for two steps deep all moves are checked for violation.

Go deeper when there are fewer moves available
Usually all moves are analyzed for three moves deep and at least the best five moves four steps deep; with the connection rule turned off four moves deep and the best five moves one step deeper. If the number of moves really to be analyzed becomes three or less I increase the search depth one extra step. There is a small risk you will spend more time than wanted; on average it does not harm too much because of the limited amount of moves.
The idea behind this is that if you are in danger you have a small number of moves available. On the other hand your opponent has a lot of moves available! By analyzing one step deeper you are able to find the exit (to a normal safe position) much more easily than your opponent can avoid it. Of course this decision does not save time; it consumes time.

Go deeper when check is given
While analyzing the search tree a lot of check situations will be discovered by both players (in your own analysis). In such a situation I perform a call to **alfabeta_'othercolor'()** with the same depth as called. The reason for this is explained in the last paragraph. This decision does not save time; it consumes time.

Make it faster by going deeper!
If the desired depth is achieved and the queen is given check I have decided to make an extra call to **alfabeta_'othercolor'()**. Due to the deeper analysis it seems that a better evaluation is achieved and that sooner low or high fails occur due to alfabeta pruning. In this particular situation no time is wasted; it saves time. I have measured an improvement in time of about 20 %. Interesting, isn't it?

Make the search window as small as possible
I have used a window of integers in the range of -25 to 25. I have narrowed it as much as possible to obtain fast low and high fails with alfabeta pruning as soon as possible. By narrowing it you lose a lot of details in the evaluation but that is not really a

disadvantage because it has been as simple as it could be. Furthermore the number of iterations in the MTD(f) routine decreases as well. So the search depth increases heavily because of these two arguments.

How to make an adequate evaluation function?

For Caïssa I have made it simple and fast. For a situation in which a white win is found an evaluation between 22 and 24 is returned. The value 24 is used for a close win and 22 is used for a deep win. For a black win the evaluation lies between -24 and -22. For the other cases the following formula is used:

Evaluation = Sum(value for each piece) + Number(white moves) - Number(black moves)[**]

white rook	**48**
white bishop	**30**
white knight	**24**
black rook	**- 48**
black bishop	**- 30**
black knight	**- 24**

The number of moves is returned by the generator. The last two results are used; for one of them the result from **alfabeta_'color'()** from the calling depth is used. By using this evaluation the capture of a piece is more important than having to play more possible moves than your opponent.

How to restrict the returned evaluation scores?
The evaluations calculated by the formula were not returned directly. They had to be made smaller first so I have divided them by 6 or by 12 and have shifted them a little to fit to each other. If the evaluation was small (between -24 and 24) I have divided them by 6. In the beginning of the game the neutral evaluation is zero and a division by 6 preserved enough detail for a useful evaluation. Once a piece has been captured the initial evaluation turns out to be less than -24 or bigger than 24. For those cases I divide by 12. I have limited the return values obtained this way between -21 and 21.

Not to stop in a check situation!
In a situation where check is given an other **alfabeta_'color'()** is performed. The evaluation is in favor for the last player that has not been given check. If you are given check the number of possible moves is always very small. So the evaluation is searching for check situations and that turns out to be effective.

[**] I have had several discussions about this evaluation with one of my students. Joost Michielsen is a former Dutch champion in chess; he can tell you anything about Fritz 6! I showed him Caïssa and he got interested in playing. He could show me the weak points in my program. The evaluation in which our ideas were combined seemed to function best.

How to store and play moves?

The storage of the pieces and the tiles
To store the pieces and the tiles I have used some defines and two one-dimensional arrays:

```
#define BQ -4
#define BR -3
#define BK -2
#define BB -1
#define WQ 4
#define WR 3
#define WK 2
#define WB 1
int board[49] = { 0,0,0,0,0,0,0 , 0,0,0,BB,BQ,BK,0 , 0,0,0,0,BR,0,0 , ...};
int tiles[49] = { 1,1,1,1,1,1,1 , 1,1,1,1,1,1,1 , 1,1,1,1,1,1,1, ...};
```

To make the generator efficient the separated positions of the pieces are stored as well. Its value could be CAPTURED as well.

```
#define CAPTURED 49
int pos_bb=10;
int pos_bq=11;
int pos_bk=12;
int pos_br=18;
int pos_wr=30;
int pos_wk=36;
int pos_wq=37;
int pos_wb=38;
```

And at last I administrate the number of pieces and the number of tiles. The number of pieces is used to make the decision to perform a normal alfabeta analysis or an endgame analysis (which never occurred in the Codecup). Furthermore this variable is used to calculate the evaluation for a stalemate situation. The number of tiles is used to check for violation of the connection rule.

```
int number_of_pieces=6; // both queens are not counted
int number_of_tiles=49;
```

Stack and stack pointer explained
For Caïssa I have used an array of integers to store the generated moves. The array is declared globally for fast access. All generated moves are stored on this stack. The stackpointer sp is used to store a generated move.

```
int stack[500][4];
int sp;
```

For Caïssa a stack of 500 was sufficient. The average number of moves is about 15. If a depth of twelve is achieved about 150 moves are stored on the stack.
A move is stored using 4 fields.

In the first field the sort of move is stored; there are five of them:

1. normal piece move without the displacement of a tile;
2. normal piece move with the displacement of a tile;
3. normal move of swapping a piece;
4. queen move with a capture;
5. regular queen displacement.

In the second field the origin value is stored and in the third the target value. The fourth field is used to store evaluation values obtained by the MTD(f) function.

The do & undo function

The **do_move_white()** and **do_move_black()** functions are called with the index of the position of the move on the stack. What it does is trivial:

1. it moves the piece to its new position by administrating the array board;
2. it administrates the array tiles;
3. it administrates the position of the pieces;
4. if needed the number of pieces and the number of tiles are adapted;
5. in case of a capture the captured piece is returned.

The functions **undo_move_white()** and **undo_move_black()** do the opposite.

How to generate moves?

The main structure of the generator of moves

To make the generator of moves fast, it does some research first and finally starts searching for moves. This is visualized in the following scheme[††]:

<table>
<tr><td colspan="3">First it is investigated how many times check is given.
If this is done by one piece only the position of the piece is stored;
also in which direction it is giving check is stored.
If the check giving piece is the knight, this is stored separately.
If one of your own pieces is pinned its direction is stored.</td></tr>
<tr><td>if not is given check:</td><td>if check is given once:</td><td>if is given check more than once:</td></tr>
<tr><td>search queen moves</td><td>search queen moves limited version</td><td rowspan="4">only search for queen moves</td></tr>
<tr><td>search rook moves</td><td>search rook moves limited version</td></tr>
<tr><td>search bishop moves</td><td>search bishop moves limited version</td></tr>
<tr><td>search knight moves</td><td>search knight moves limited version</td></tr>
</table>

If a piece is pinned you only have a limited amount of legal moves available. If you investigate all different situations separately these moves are very fast to find because just a few checks are needed to calculate if a move is legal or not. To accomplish this it is necessary to store the direction in which the piece is pinned. A full explanation is given with examples in this chapter.

[††] This scheme was designed by Frank van Gemeren, Bas van den Aardweg and myself. It took more than 80 hours of discussion time, programming and debugging. I finally succeeded in making it error free. Frank and Bas succeeded almost; in the final competition just a single error appeared. The programming and debugging was done completely separately. We played several test matches to search for errors in our generators. That cooperation was very useful.

Using pointers in look-up tables

I have used look-up tables in the format of pointers to an integer to find moves. The board itself was declared as:

0	1	2	3	4	5	6
7	8	9	BB	BQ	BK	13
14	15	16	17	BR	19	20
21	22	23	24	25	26	27
28	29	WR	31	32	33	34
35	WK	WQ	WB	39	40	41
42	43	44	45	46	47	48

The position pictures are copyrighted by Ed van Zon; for more information visit the website www.mindsports.net.

```
int board[49] = { 0,0,0,0,0,0,0 , 0,0,0,BB,BQ,BK,0 , 0,0,0,0,BR,0,0 , ...};
```

To achieve a series of possible moves for the queen in the case it only moves like a king, the following look-up table was used:

```
const int *p_kingmoves[49][9]=
{
   {board+1, board+8, board+7, NULL},
   {board+2, board+9, board+8, board+7, board, NULL},
   ...
};
```

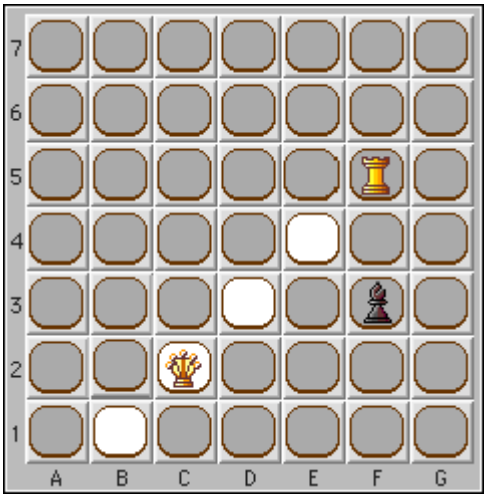
The first line indicates that you can move from field 0 to the fields 1, 8 and 7. The NULL indicates that there are no more moves left. The second line is for the moves from field 1. For the other piece moves similar look-up tables were used:

```
const int *p_knightmoves[49][9];
const int *p_moves_direction1[49][7]; // direction is north-east
const int *p_moves_direction2[49][7]; // direction is east
const int *p_moves_direction3[49][7]; // direction is south-east
const int *p_moves_direction4[49][7]; // direction is south
const int *p_moves_direction5[49][7]; // direction is south-west
const int *p_moves_direction6[49][7]; // direction is west
const int *p_moves_direction7[49][7]; // direction is north-west
const int *p_moves_direction8[49][7]; // direction is north
```

To obtain the rook moves the even direction tables only were used. For the bishop moves the odd direction tables were used. For the normal queen moves all eight direction tables were used.

Using pointers to pointers in look-up tables

I also used another type of look-up tables. While searching for moves you have to investigate the situation if the queen is given check after a displacement. This is solved by using an array of pointers to an array of pointers to an integer.



0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	WR	20
21	22	23	24	25	26	27
28	29	30	31	32	BB	34
35	36	WQ	38	39	40	41
42	43	44	45	46	47	48

Suppose you want to play F5-F3. This is not possible because the white queen would be given check by the black bishop. It would be nice to have information about the white queen (field 37) and the field where the bishop (field 19) arrives after the rook move. This information is available in an array:

```
const int **watched_by_bishop[49][49];
```

Let us look at the array element [19][37]:

```
watched_by_bishop[19][37]=p_moves_direction5[19]
```

Its value does not contain the NULL pointer. If so you're sure that field 37 can be seen by a bishop at field 19. The actual direction is direction 5 (that is south-west of field 19).
By using a while loop, you can check easily if there are pieces in between:

```
int **p;
p=watched_by_bishop[19][37];
if (p==NULL) return bishop_cannot_see_white_queen;
while (*p) // in this case always true because the field with the queen must be found!
{
   if (**p) // check if the field is occupied
   {
      if (**p==WQ) return no_piece_in_between; // you have ended on the white queen
      else return piece_found_in_between; // another piece is found
   }
   ++p;
}
```

The pointer p is initialized with the value p_moves_direction5[19], so the pointer *p starts with: p_moves_direction5[19][0] = board + 25. After performing the operator++ on the pointer p, it becomes p_moves_direction5[19] + 1. The new value of the pointer *p now is: p_moves_direction5[19][1] = board + 31.

If all the fields were empty the operator++ would do this for *p in the while loop:

p_moves_direction5[19][0]:	**board + 25**
p_moves_direction5[19][1]:	**board + 31**
p_moves_direction5[19][2]:	**board + 37**
p_moves_direction5[19][3]:	**board + 43**
p_moves_direction5[19][4]:	**NULL**

There are similar arrays for the rook and the queen:

```
const int **watched_by_rook[49][49];
const int **watched_by_queen[49][49];
```

Generating the queen moves

As pointed out in the structure of the generator of moves there are two routines doing this. If the queen is given check it has a limited movability and the array p_kingmoves is used. If it is not given check the eight direction arrays are used to generate the moves. To see how this is done in detail you should have a look in my source code‡‡.

There is one detail I would like to explain. If the white queen is given check a possibility is to let it look for a safe place for herself. Discard the next situation.

‡‡ You can find the complete source code of HIPP on my website www.vlastuin.net. Go to the Caïssa homepage and download hipp.zip. The code of the generator for white moves is found in generatw.c.

The white colored fields are forbidden for the white queen because they are attacked by the black pieces. To recognize them you need to remove the white queen temporarily while looking for safe places. The only safe place is found on D1.

Generating the rook and bishop moves in check situations

The search for the rook and the bishop moves are almost similar to each other. The basic idea about how to look for the moves is explained first. Suppose the white queen is given check once by the black bishop. We are looking for moves of the white rook:

The white rook has to search in three directions only. If looking to the west it stops at the critical field B2. You can do this by using watched_by_bishop[F5][B2]. For the same reason it will stop going north at E4. Only the eight white fields are analyzed for the white rook. Once a critical field has been found you can search for a move by checking if the rook is between the queen and the bishop. The direction in which the bishop is giving check is known. Using a pointer you can check the fields in that direction. If the rook on its new position is be found first you store the corresponding move. If the queen is found you ignore it.

Suppose we have the next situation. The bishop again is giving check.

If two moves (D4-E4 and D4-D3) are found, it is not possible to find more! In my program I stop looking for rook moves in such a situation. I always start looking east.

If the queen is given check once and your rook is pinned you can find legal moves. Only little effort is needed. Discard the next two situations.

In the left diagram the white bishop is pinned. If there is a move available for the bishop; it must search for a swap with the check giving piece. The positions of both pieces are known. The check giving piece is located on G2; the pinned bishop on E4. With the look-up table element watched_by_bishop[E4][G2] it can be tested if the white bishop can see the black rook(F3 must be empty) . If so you need to investigate if the fields located in between are empty. For that watched_by_bishop[E4][G2] is to be used. In the right diagram there is a similar situation for the white rook. In this case watched_by_rook[D2][D4] could be used to find the move.

In the last two examples it makes no difference with which piece it was swapped. If the pinned piece is a bishop and is pinned horizontally you have to be careful. You may not swap with the black bishop. The same counts for the rook that is pinned diagonally. This is shown in the next two diagrams.

In the left diagram there are no bishop moves. In the right diagram there are no rook moves available.

Generating the rook and bishop moves in non-check situations

If the rook or the bishop is pinned it is easy to find moves. Discard the next two cases.

The white rook is pinned vertically. If searching for vertical moves the rook may only move without swapping. Horizontally the rook may only swap with any piece it can find. The piece that arrives on C5 can never give check because the black rook is pinning the white rook. In the left diagram a similar situation is drawn for the white bishop: it can move in the directions north-east and south-west and it can swap pieces into the other two diagonal directions.

The rook and the bishop can be pinned into the other directions as well. In that case you have to look for other sorts of moves. Look at the two examples below.

The white rook is pinned diagonally in both diagrams. In the left diagram the rook is pinned by the black bishop. The only moves it can make are moves with swapping pieces. If the white rook is pinned by the black queen there is the danger of swapping with the black bishop. If you should swap with the black bishop you should exclude the move. Similarly a horizontally pinned white bishop may not swap with the black rook.

Generating the knight moves

The knight moves are generated in the same way as the rook and bishop moves, because the knight jumps only everything is a little bit easier. The look-up table p_knightmoves[49][9] makes it easy to find all the moves. There is an equivalent for the watched_by_rook/bishop arrays. In the watched_by_knight array the information is stored if a knight can or cannot see another field from a certain position only:

```
const int watched_by_knight[49][49]=
{
   {0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0 …},
   ….
}
```

If a knight is pinned diagonally you may not swap it with the other colored bishop. If pinned horizontally or vertically the knight may not swap it with the other colored rook. In the case the knight is pinned and the queen is given check you have to be very careful if you want to swap with the check giving piece. If the knight is pinned diagonally there are no restrictions at all. It is not possible for the knight to swap with the check giving bishop. This is shown in the diagram.

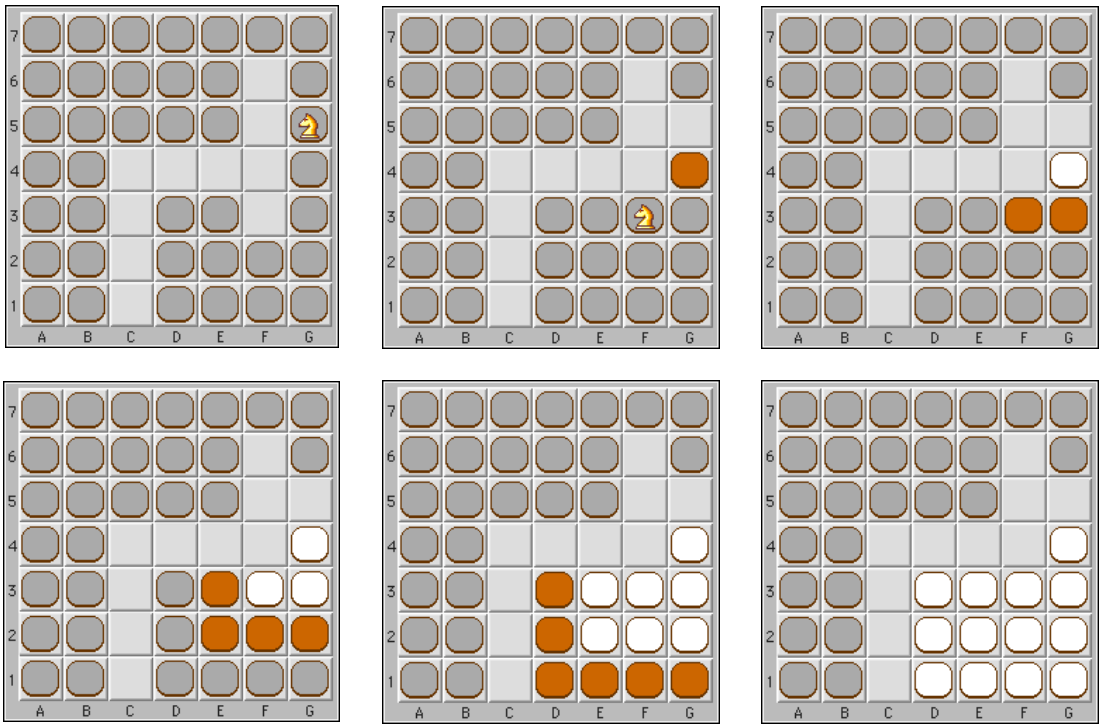
However if the knight is pinned horizontally or vertically it is possible to swap with a check giving rook! In such a case an extra check is needed. See the diagram.

Dealing with swap moves of the rook, bishop or knight

In general two things can be said about finding moves in which two pieces swap. The first is that the violation of the connection rule never occurs. So there is no need to check for it. The other remark is about putting yourself check after performing a swap. In general you have to check for that before storing a move. Mostly in the pinned cases this could be omitted; there was only one exception.

My solution for the connection rule

I have made a small routine to check for the violation of the connection rule. I used a look-up table for the neighbors of a tile. I started counting the tiles from the disappearing tile. There is a change you start counting a small amount isolated tiles! I used a while loop to search the neighbors of the neighbors until no new tiles were found; meanwhile I counted the tiles and removed them immediately afterwards. In the diagrams below such a search is visualized. For this example 5 loops were needed; in the last loop no new neighbor tiles were found.



The while loop in the source code is about ten lines. It is pretty fast but if I had used bitboards like Vincent Groenhuis and had used his technique[§§] this routine could be much faster. What he did was really smart!

[§§] For more details read the discussion on www.algoritme.nl about Caïssa.

Using an opening book

There are three big advantages if an opening book is used:

1. it is fast;
2. for the remaining moves more playing time is available;
3. the best moves according the evaluation are played.

In the diagram the initial called depths for the analysis are shown for all the moves. The numbers of depths marked with an apostrophe are read from the opening book.

number of moves	white moves	black moves
1st move	10'	9'
2nd move	9'	8'
3rd move	8'	7'
4th move	7'	5 or 6
5th move and further (without connection rule used)	5 or 6	5 or 6
endgame (with connection rule used)	4 or 5	4 or 5

It is important not to calculate the opening book moves too deeply. If the last move from the opening book is calculated too well the real time analysis cannot find the proper next move. In the Codecup the following situation appeared twice. White plays the fourth move C3-G3 from the opening book. Black makes a mistake and captures the rook! White wins in three moves because the next moves were found correctly.

My opening book contains 2226 white moves and 7804 black moves. The opening book for the second white moves is based on the first white move and so forth. The same holds good for the black moves. I have stored the moves in an array with two fields. In the first field I stored a transcription of a position. In the second field I stored the move for that position. The transcription is not unique for a certain position. I had to check for the same transcriptions with different moves. These are called *collisions*; I have removed about 15 of them. I have sorted the array for the positions so that a fast binary search could be performed to find the corresponding moves. If a position was not found I stopped reading the opening book.

It is possible to store a position in 95 bits; so I could use three 32 bit integers a, b and c to store the position. The transcription was calculated by using the *exclusive or* operator: a^b^c. If the algorithm for the calculation of the integers a, b and c is chosen well the number of collisions is small.

Bibliography and websites

[1.] Aske Plaat
Research Re: search & Re-search,
PhD Thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, June 20, 1996.

[2.] http://www.xs4all.nl/~aske/
This is the personal website of Aske Plaat.

[3.] http://digilander.libero.it/gargamellachess/papers.htm
On this Italian website a lot of scientific papers are found about computer chess. All the papers are downloadable in pdf format.

[4.] http://www.vlastuin.net
This is my personal website. On this site I have a Caïssa homepage. You can download the source code of a fully documented version of HIPP and this paper as a pdf document.

[5.] http://www.codecup.nl
This is the website of Kim Schrijvers and Joris van Rantwijk. I participated with HIPP in the first contest and won it in a field of 30 participants.

[6.] http://www.informaticaolympiade.nl
This is the official Dutch Informatics Olympiad website. Willem van der Vegt is the webmaster of this inspiring site. The game Caïssa was one of the problems to be solved for the students in the first round. The *Nederlandse Informatica Olympiade* is an Olympiad for 12- to 18-year-old Dutch students.

[7.] http://www.algoritme.nl
This is the unofficial Dutch Informatics Olympiad website. Wilmer van der Gaast is the webmaster of this excellent site. Wilmer is an ex-participant of the Dutch Informatics Olympiad. The site actually functions as a newsgroup. A lot of contestants of the Codecup and the Dutch Informatics Olympiad had discussions and organized test rounds for Caïssa on this site.