Ideas regarding chess endgames

Jeroen Laros

December 13, 2003

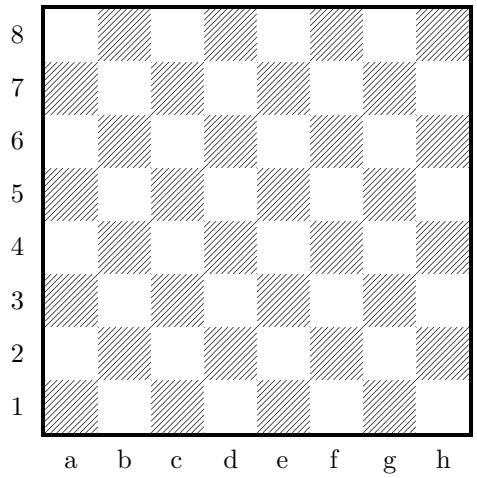
Contents

1 Introduction

Because chess endgames [2] have few pieces on the board it is possible to exhaust all possibilities and pick the best move. That is why not much attention is given to endgames, especially the simple ones. From another point of view there are still some things of interest. For example: what is the average mate-depth when using a certain strategy? Is there an optimal strategy and is it as good as the brute-force solution? How do we find such a strategy? This is a project for the Leiden Institute of Advanced Computer Science, Leiden University as part of the study Computer Science. This particular project is corrected and guided by Walter A. Kosters.

2 The game of chess

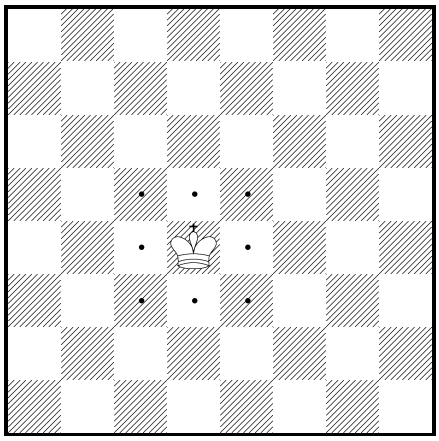
Chess is a two-player game played on an 8×8 board of which the columns are numbered a to h and the rows are numbered 1 to 8. So the top-left square is denoted with a8 and the bottom-right one is h1.



Each player begins with 16 pieces, of a particular colour, black or white. In this document we only use a few pieces, so we will only discuss these particular pieces and only the moves they can do that are relevant for this article will be discussed (no en-passant, promotion or rochade that is).

2.1 The king

The king ♔ is the most important piece on the board. It may move in the following ways (the possible moves are marked with dots):

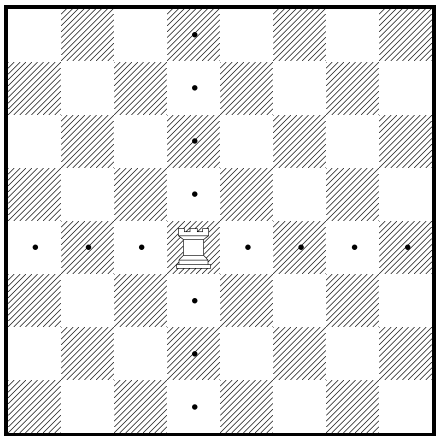


If the king is threatened with capture the king is in check. A player may not leave his king in check. This means that if your king is in check, you must find a way not to leave it in check, nor may you put your own king in check. If a player is unable to make such a move, the king is mated and the other player wins the game.
If a situation arises where the king is not in check and there are no more pieces (of the same colour) on the board that can move, the game ends in a stalemate; this is a draw. There are three other situations in which the game is drawn:
- If there is no mating material present for both sides.
- If in the last 50 moves nothing was captured and no pawn was moved.
- If there has been a threefold repetition in moves.

For simplicity only the 50-moves and stalemate rules are used. We also leave abundant pieces off the board (sometimes even one of the kings).

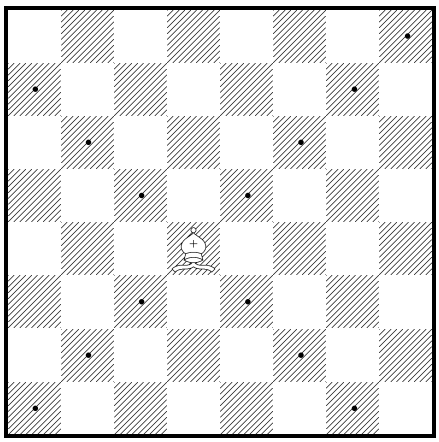
2.2 The rook

The rook ♖ may move as many squares as it can horizontally or vertically, without jumping over other pieces:



2.3 The bishop

The bishop ♗ moves like a rook, but always diagonally:

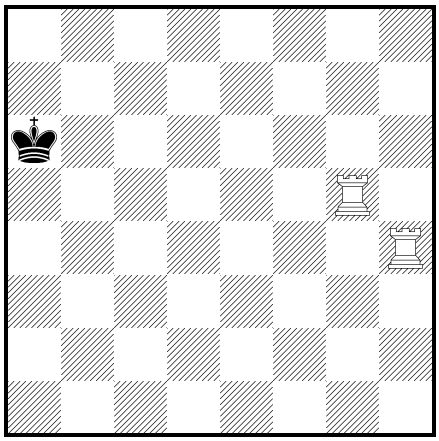


Note that because the bishop is bound to either white or black squares, it can only reach half of the board.

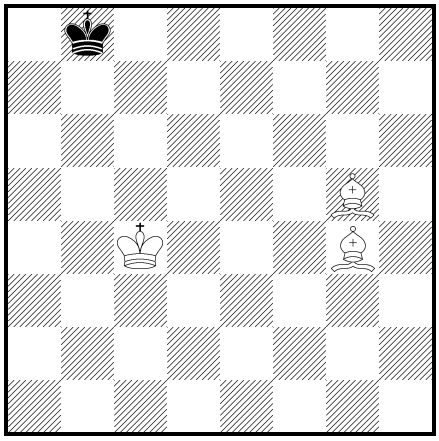
3 What is a strategy?

When you teach someone to play chess, you usually start with endgames because they are simple and with some simple rules you can point out the possibilities of

the pieces. Because you want your student to understand how the game works, you will try to give as few rules as possible to improve the position at each given situation. It will most probably not be the best move, but for educational reasons it will be a good one.
For two rooks something like this would be educational:

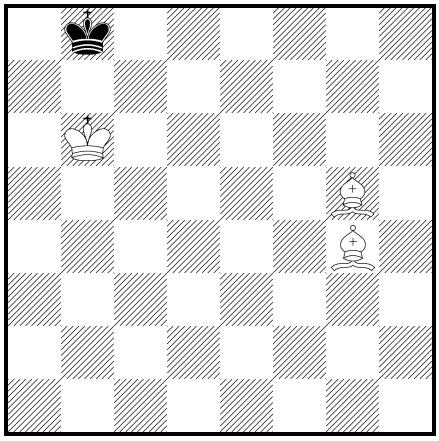


Move the rook from h4 to h6, then rook g5 to g7 and so on.
For two bishops something like this:



When the king steps of the b8-h2 diagonal, move the bishop from g5 to f4. Use your king to take away the squares on this diagonal.
Now, if you have a couple of these rule-sets you can combine them to make a mating attack. The examples above illustrate a way to force the enemy king to a position in which he can be mated. If you keep following the rules in the first

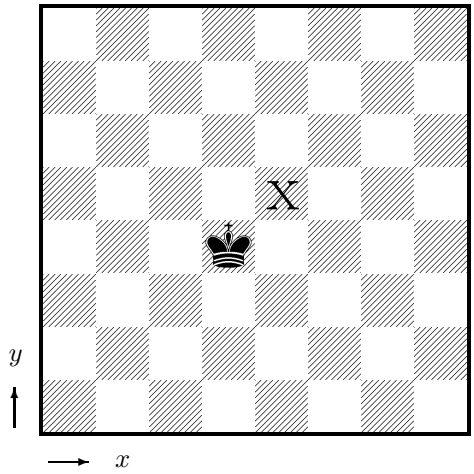
example you will mate the king. In the second example this is not the case. You will have to change the rules of the strategy at a certain time:



In this position you must give check with the bishops (♗g4 and ♗g5) to mate. The combination of some rule-sets and knowing when to apply which rule-set is what we call a strategy.

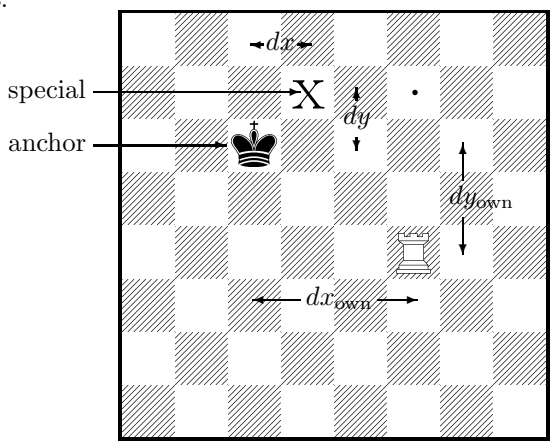
4 Idea

The main idea is for each piece to have a *special square*. This special square is a square relative to another piece, which we shall call the *anchor piece*.



It is represented by an integer pair (dx, dy) where dx is the difference in x-

coordinates and dy the difference in y-coordinates. In the picture above the special square is denoted with an "X" and the integer pair is: (1, 1). This special square can be the exact position of the anchor piece (0, 0) and the anchor piece does not have to be a piece of the opposite colour (defence). The main idea is to cover (be able to go to that square in one move) that square. In order to get things to work, defining a special square is not enough, because many positions are roughly equal. To solve this, a set of flags is implemented in order to find out how the special square should be interpreted; these are called *constraints*.



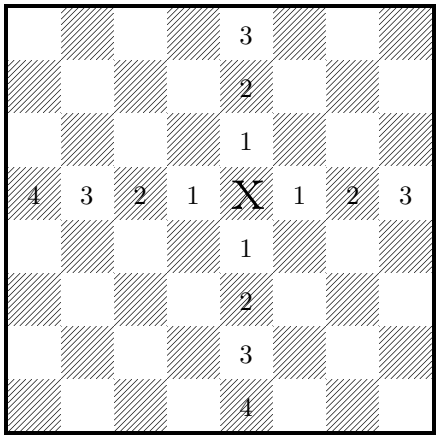
The current position of a piece with respect to the anchor piece is denoted by $(dx_{\text{own}},\ dy_{\text{own}})$. The constraints can be represented by a pair of booleans in which a 1 stands for a hard constraint and a 0 stands for a soft constraint. If for example the constraints are (1, 0) then dx is a hard constraint and dy is soft.

4.1 Constraints

Let us consider an example. The anchor piece is the enemy king, the special square is (1, 1) and the constraints are (0, 1). This means that the square at position (1, 1) relative to the enemy king is the objective. The 0 in the constraints set means that the dx-value is a soft constraint, the relative position should be larger. The 1 in the constraints set means that the dy-value is a hard constraint. Note that the soft constraint is more important than the hard constraint. If the soft constraint is not met, the piece is (probably) under attack.
In order to use "slow" pieces like a pawn, king or knight the hard constraint must reflect the badness of the position, so a simple 0 or 1 will not suffice. That is why we use Δ values instead, with $\Delta_x = \|dx_{\text{own}} - dx\|$ and $\Delta_y = \|dy_{\text{own}} - dy\|$.

4.2 The hard constraint

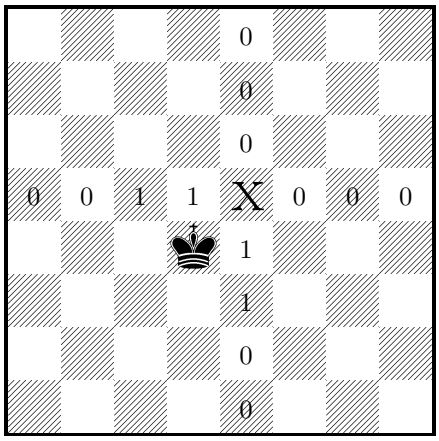
The reason this type of constraint is called "hard" is because originally it had only two realizations: good or bad. This works fine for rooks and bishops, but not for knights and kings, because they may be too far away to make a good move.



In this picture we see the penalties for all positions on the board when we use two hard constraints, the special square is the optimum and the numbers indicate the penalty of other locations. The hard constraint is used primarily for attack. We can easily extrapolate the values for the other squares by adding the x- and y-values.

4.3 The soft constraint

The soft constraint is more difficult to explain. That is because this type of constraint is used for defence. The idea behind this is that we keep away from the special square. Because we actually need to keep away from the anchor piece the special square is interpreted as a corner of a "keep away" rectangle. Note that this rectangle only exists when using two soft constraints.



The value of the special square itself is also 1. And because we regard defence as more important than attack, we multiply the values of the soft constraint by at least the maximum of the hard constraint ($x^{\max}$ or $y^{\max}$). Note that these maximums are loosely defined here. An arbitrary large value will suffice (15 for example). By doing this we will always make sure our piece is safe first by putting a bias on the soft constraint. Again we can extrapolate the rest of the values by addition.

4.4 Combining the constraints

If we now define our own relative position to the anchor piece as dx_{own} and dy_{own}, we can make two tables in which we can look up the value of a position for each type of constraint:

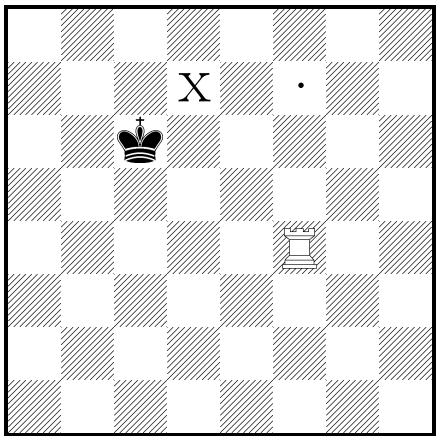
The soft constraint (0)

$\lvert dx_{\text{own}}\rvert > \lvert dx\rvert$	$\lvert dx_{\text{own}}\rvert \leq \lvert dx\rvert$	$\lvert dy_{\text{own}}\rvert > \lvert dy\rvert$	$\lvert dy_{\text{own}}\rvert \leq \lvert dy\rvert$
$+0$	$+x^{\max}$	$+0$	$+y^{\max}$

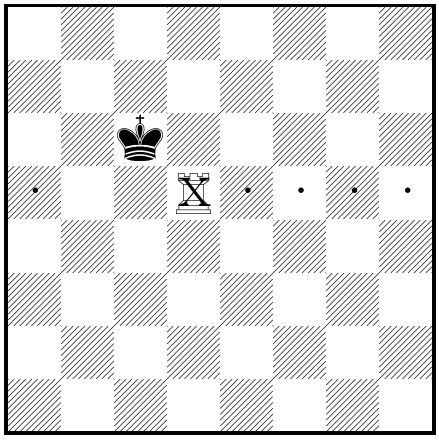
The hard constraint (1)

$dx_{\text{own}} \neq dx$	$dy_{\text{own}} \neq dy$
$+\Delta_x$	$+\Delta_y$

We add the values in the tables to the default value 0. If the position of our piece is "perfect" it will have value 0. If its position is "bad" it will have a high value. Let us look at an example:



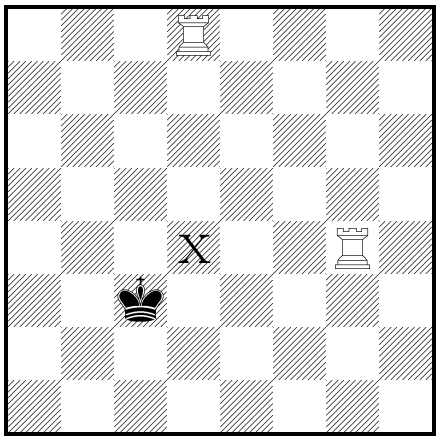
In this position the square d7 is the special square, (dx, dy) is (1, 1), the constraints are (0, 1) and (dx_{own}, dy_{own}) is (3, –2). The current value of the rook is 3 (+0 for the soft constraint and +3 for the hard constraint). Because the soft constraint is met, the rook will move to f7 to adhere to the hard constraint and improve its position. After this move the value of the rook will be 0. Another example:



Here (dx, dy) is (1, –1), the constraints are (0, 1), and (dx_{own}, dy_{own}) also happens to be (1, –1). In this position the soft constraint is not met, this means that the rook is too close to the anchor piece and it is probably under attack. So the rook must move away from the king horizontally. Why not vertically one might wonder; well, this is because the hard constraint is met and can only be maintained by a horizontal move. Moreover, the rook must go to column a, e,

f, g or h to satisfy the soft constraint.

Another example:

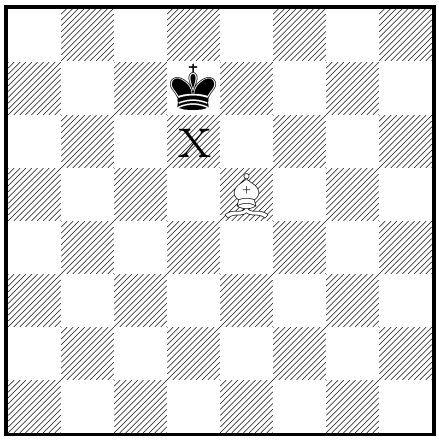


Let us assume that both rooks have the same special square, but the constraint sets are different: (0, 1) for ♖g4 vs. (1, 0) for ♖d8. The position above is ideal and cannot be improved, a move like ♖h4 will not make things worse. This is called zugzwang ⊙, leaving your opponent in the same position but now it is his move. Because he only has a king left, moving it will deteriorate his position.

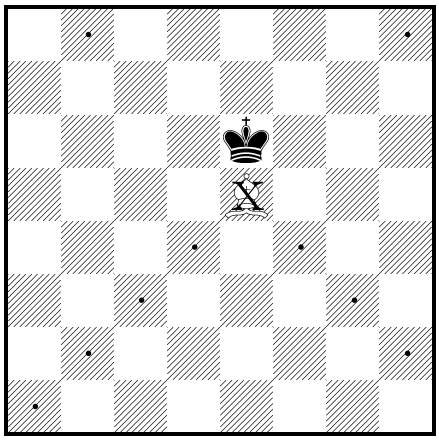
With this approach (and the right parameters) it is possible to make sure that each move either improves or holds the position. However, it is assumed that a certain global state is true (in this example: at most one rook is under attack). Because this is not always the case, the parameters should be able to change. To do this, a kind of global state must be implemented, and the possibility to jump to a different set of parameters if a certain global state is reached.

5 Details

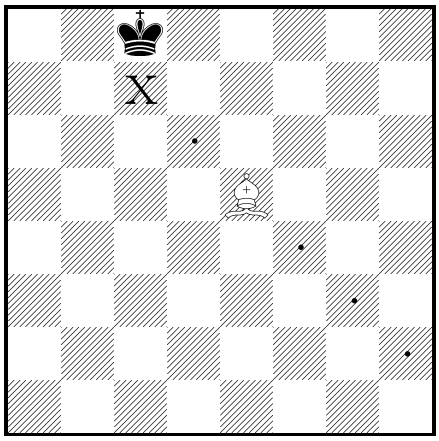
It would seem that this approach would only work with rooks, but it can also be used for bishops:



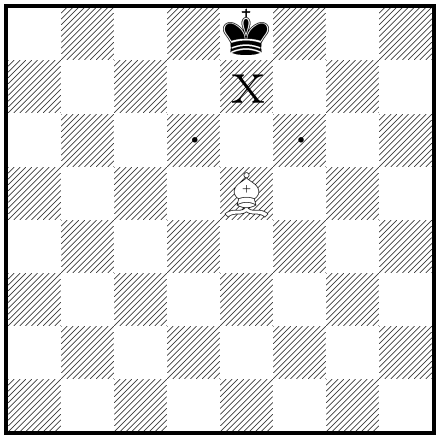
Let us assume that the bishop has (1, 0) as constraint set. If the king now attacks, the bishop will flee:



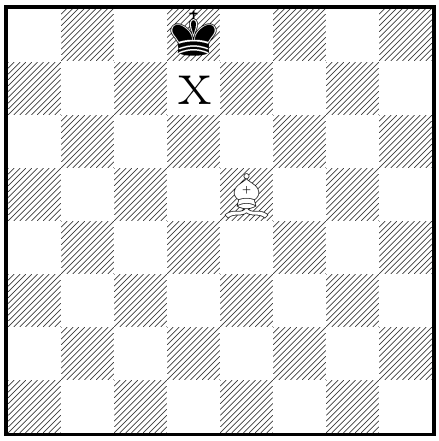
If however the king goes to the back row, the bishop will hold its position by doing a null-move (a move that does not really matter):



And if the king decides to walk off the endangered diagonal:

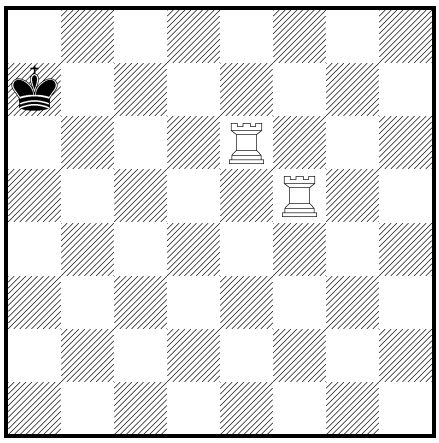


The king also could “change” the colour of the square it is standing on:

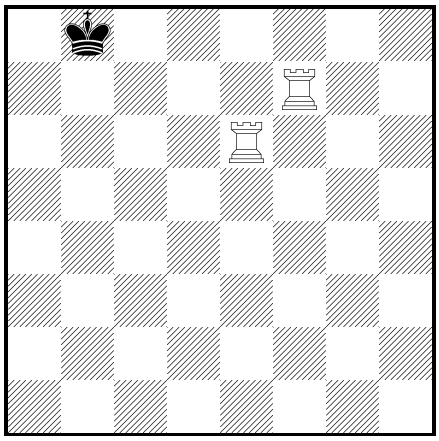


In this case the bishop will stay where it is, as long as the soft constraint is met, otherwise it will flee.

For somewhat more complex strategies the roles of pieces must sometimes be swapped.



Let us assume that the position for the ♖e6 is ideal, and that the ♖f5 must give check by moving two squares up.



If we continue with the same strategy, the ♖e6 must move one square up, but better is to use ♖e6 to give check. Simply swap the roles of the rooks.

6 Implementation

In this section we will use the word program for the parameters and jumps to other parameters, the input for the chess program.
A way to implement all this is to code the program in the genes of a *genetic algorithm* [3]. This is not really straightforward, as the program is not fixed in size. Our choice for a genetic algorithm is because such an algorithm by definition uses a translation between the genetic code and what it is used for. Because some parameters in our program are much more important than others we might want to code some parameters in duplo or more. For example: we probably want to give the coordinates of the special square more chance to change than the jumps.
Another thing to worry about is how to recognise a global state. A kind of hashing function must be used for this. This function must take into account the kind of pieces, their state and the number of pieces.
In this document we will only look at the programs containing one strategy, the so-called building blocks.

6.1 Genetic algorithm

We use a genetic algorithm (GA, see [3]) to explore the possibilities. A genetic algorithm is basically an algorithm in which candidate solutions are given a grade (this is done by the so-called *evaluation function*), combined with each other, altered slightly, and graded again until the grade is high enough. We use

a generational model where the children replace the parents. In this particular algorithm we take about five candidate solutions from a population of about 50 and we alter these solutions with both mutation and crossover.
As the genes in our genetic algorithm are in fact strategies, or small programs, the term *genetic programming* is perhaps more suitable. For more about genetic programming, see [1]. The reader is referred to [4] for more on artificial intelligence in general.

6.2 Crossover

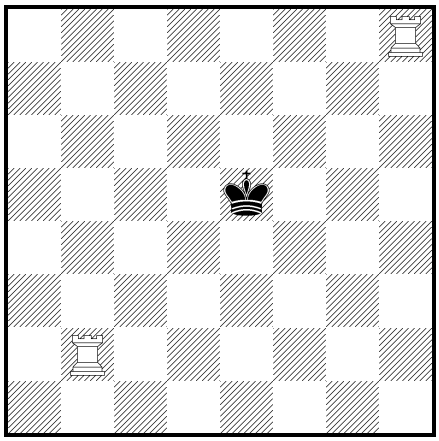
We use *crossover* to combine individuals. In this crossover scheme we see the values in a strategy as a gene. So if we do a crossover between the individuals (–2, 1, 0, 1) and (2, –3, 1, 1) and we take a random mask like (1, 0, 0, 1), which means which gene we will take from which parent, we get two offsprings: (–2, –3, 1, 1) and (2, 1, 0, 1). We can also do a crossover on a higher level, but there are no higher levels implemented at this time.

6.3 Evaluation

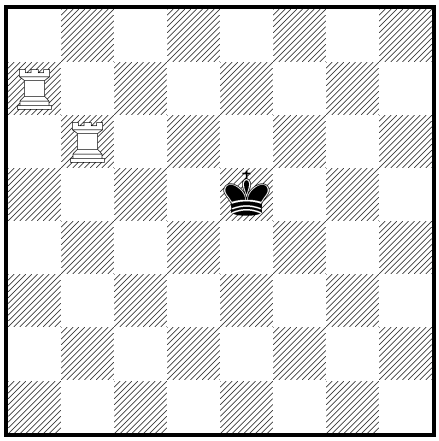
The evaluation function simulates a game of chess against a random playing opponent. It does so by allowing both players 50 moves, after which a draw is declared. If a piece is captured, the game is lost. A stalemate or mate wins. A win gives the winner 100 – the number of moves needed points, on a draw 25 points are given and 0 points for a loss.

7 First experiments

At first we tested one particular situation (note that no white king is present):



At this stage the anchor piece is always the king (for both rooks) and no changes in strategies are allowed yet. Still some interesting things happened. In the first tests we treated stalemate as a normal mate. What happened was that several strategies were found, most of them were stalemate techniques, but there were real mating techniques amongst them. At first we assumed no real mating technique were to be found, but it is possible to mate without changing the roles of the rooks, although this is not a very fast way to mate a king. Let us look at the main idea behind this form of attack:



Now, instead of moving the ♖a7 to a5 we move it to a6. The next time we move the ♖b6 to b5. Eventually the king will get mated on row 1. Note that the rooks will not be captured if the king walks towards them, because of the soft constraint they will keep well away from the king.

7.1 Mutation only

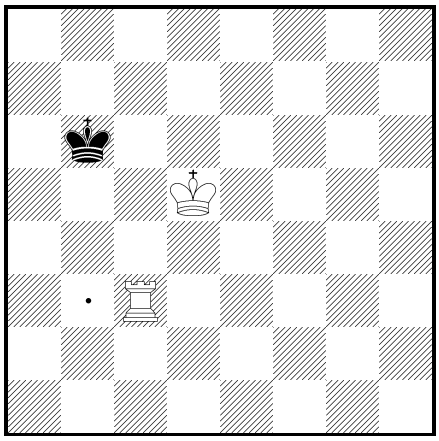
These results looked very promising, but at this stage no crossover was implemented yet, nor was the *elitist selection*, which keeps the best individuals. Sometimes the algorithm did not find the global optimum, but stayed in a local optimum. To circumvent this problem we first implemented the elitist selection, which was of great help. The algorithm became a lot faster, but it did not became more reliable. In order to make it more reliable we implemented the crossover. Note that by using crossover we do a more global search. This seems to work, the algorithm always terminates, but extensive testing still has to be done.

7.2 Mating

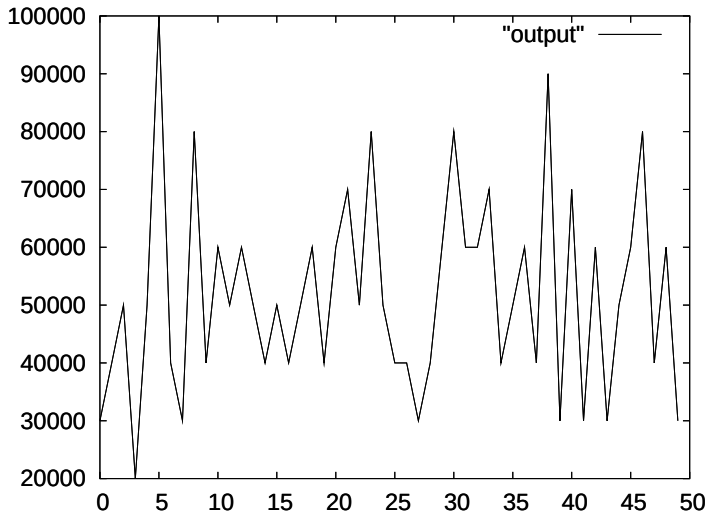
After implementing the mate in our chess program we could test the mating capabilities of this approach. This seemed to be a lot more difficult than finding a stalemate. We concluded that the evaluation function was not ready for this task, so we adjusted this function to find mating attacks more easily. We decided that everything can be seen as a loss, except the mate. The theory behind this is that if you give points for draws, the algorithm could play for a draw without noticing it could win. This approach speeded up the process a great deal. Obviously a stalemate was a local optimum which was very hard to get out of. This is not surprising, as a stalemate technique is very different from a mating technique.

8 Results

Because at this time only two pieces are implemented, just a few tests could be done. Let us look at the first one: ♔ against ♖ ♖. One of the strategies the program came up with was (2, –1), (0, 1) for the first rook and (2, 0), (0, 1) for the second one. This means that one rook is used to block the anchor piece and the other rook is used to give check. If we continue to repeat this technique the anchor piece will be mated eventually on the edge of the board. As expected, eight of these techniques were found, one for each border times two because we got two rooks.
Another test we did was: ♔ against ♔ ♖. Note that we cannot find a perfect strategy for this one, because we have to make a change of strategies. To be more specific: if the ideal position we are trying to realise is not one in which the anchor piece is in check, then we did not find a mating technique. However, we found a strategy which mates the anchor piece about 60% of the times. The main idea behind this is putting the enemy king in check with the rook and hoping that it will walk to the “wrong” side:



The rook will move to b3 and if the king goes to the a-column our own king can move to get the same type of position shown above. With a bit of luck the black king will get mated on a8. Because our king prevents the escape of the black king for two of the three squares the chance is high that we mate the enemy as long as it plays randomly.
The program has a mean runtime of about 45 seconds on an Intel Pentium IV at 1.7 GHz. That is, when an ideal strategy can be found. If this is not the case finding an acceptable strategy can take over two minutes. Here is a graph which displays the number of strategies tested until an acceptable strategy was found:



About 1600 strategies are evaluated per second.

9 Conclusion and further research

The results look promising. We found acceptable solutions for the given problems. It might be nice to combine the solutions into a program which consists of a couple of strategies in order to do more complicated things like playing the king versus king, rook endgame (which should always be won, not in 60% of the cases). We could also try the really complicated endgames with this approach (like king versus king, bishop, knight).
It might also be a good idea to give a method to determine the special square, instead of just giving it. This will make sure that the enemy piece is not chased around the board, but that the strategy adapts to the global position. Another nice idea might be to include some knowledge of the freedom of the enemy king. The ability to load and save strategies would be another nice feature which would save a lot of time.

References

[1] Banzhaf, W. Nordin, P. Keller, R. E. and Francone, F. D. Genetic Programming, Morgan Kaufmann Publishers, Inc., 1997.

[2] Marsland, T. A. Computers, chess and cognition, Springer Verlag, 1990.

[3] Michalewicz, Z. Genetic Algorithms + Data Structures = Evolution Programs, third edition, Springer Verlag, 1996.

[4] Russell, S. and Norvig, P. Artificial Intelligence, second edition, Pearson Education, Inc., 2003.