Yugoslav Journal of Operations Research
17 (2007), Number 2, 257-274
DOI: 10.2298/YUJOR0702257V

# AN ALGORITHM FOR THE DETECTION OF MOVE REPETITION WITHOUT THE USE OF HASH-KEYS

Vladan VUČKOVIĆ, Đorđe VIDANOVIĆ

*Faculty of Electronic Engineering*
*Faculty of Philosophy*
*University of Niš, Serbia*

Received: July 2006 / Accepted: September 2007

**Abstract:** This paper addresses the theoretical and practical aspects of an important problem in computer chess programming – the problem of draw detection in cases of position repetition. The standard approach used in the majority of computer chess programs is hash-oriented. This method is sufficient in most cases, as the Zobrist keys are already present due to the systemic positional hashing, so that they need not be computed anew for the purpose of draw detection. The new type of the algorithm that we have developed solves the problem of draw detection in cases when Zobrist keys are not used in the program, i.e. in cases when the memory is not hashed.

**Keywords:** Theory of games, algorithms in computer chess, repetition detection.

## 1. INTRODUCTION

The solution to the problem of move repetition detection is important for the creation of a computer chess playing program that follows and implements all official rules of chess in its search [2], [5]. Namely, in accordance with the official rules of chess there are two major types of positions in which a draw is proclaimed: (a) when the position in which the same side to move has been repeated three times in the course of a game and (b) when no capture has been made or no pawns have been moved during the last fifty consecutive moves. Having in mind that condition (b) can be implemented comparatively easily by using the data from the move generator, we will focus more closely on condition (a).

The essence of the draw detection rule aims to prevent "endless" games and to offer a possibility for a game to progress by means of non-reversible moves such as pawn movement or piece exchange and captures. Naturally, in a game of average length a

relatively high percentage of moves is due to maneuverability of piece moves that lead to progress in games through a genuine transformation.

Bearing in mind the nature of the game of chess it is not merely sufficient to investigate only positions in the course of the tree search generation (decision making) [7],[8], but also to consider the game set-up prior to the actual position - the so-called "game history".

The standard approach used in the majority of computer chess programs for solving the problem of draw detection is hash-oriented. If the program in its search employs hash memory, it transforms the current position by generating 64-bit Zobrist keys [4],[6] so that the 64-bit equivalents of the positions are compared. As we said, this method is sufficient in most cases, as the Zobrist keys are already present due to the systemic positional hashing.

The new type of the algorithm that we have developed solves the problem of draw detection in cases when Zobrist keys are not used in the program, or logic is in hardware where large memory hash portions are not suitable. Namely, in very sharp and tactically oriented chess programs the efficiency of *Qsearch* is essential, so that it often focuses on fast capture/check/promotion alpha-beta search, without any hashing. This is so as it has been pointed out that the process of random generation of Zobrist keys as well as accessing slow operating memory can be critical in sharp tactical positions. However, even in such programs one must implement some kind of draw detection so that *Qsearch* could perform well and correctly.

The new algorithm is based on using variant strings generated by the search algorithm during the tree expansion in decision-making. As far as efficiency is concerned, if one takes the time needed for the generation and comparison of 64-bit keys, our algorithm performs just as efficiently as the standard methods. A detailed and precise comparison is difficult as the speed of our algorithm depends much on the current position so that only statistical parameters could offer a clearer indicator. However, our algorithm appears to be superior in positions with just a few pieces and in long checking sequences (endgames).

This paper has a listing of the procedure in *Pascal* that illustrates the details of the algorithm. The equivalent assembly coded routine is part of the AXON chess program developed by the authors. AXON, in its current incarnation, plays chess at master strength (ca. 2450-2480 Elo, based on both AXON vs computer programs and AXON vs human masters in over 3000 games altogether).

Thus we intend to offer a theoretical and practical solution to the issues we mentioned. There will be several sections dealing with various aspects of draw detection. After this initial exposition, our next section will describe the classical approach to draw detection based on the comparison of matrices. The third section will present a new treatment that employs variant strings. In the fourth section we plan to lay out a blueprint for a solving algorithm. The fifth part of the article will include a description of a routine written in assembly code implementing the discussed algorithm, while the concluding section will deal with some experimental data relevant to the algorithm as implemented in the AXON chess program. We will compare the behavior of the program with and without the implemented procedure for draw detection.

## 2. MATRIX APPROACH TO THE SOLUTION OF THE PROBLEM

In the process of recognition of draw detection and recurrently identical positions chess programs employ the same memory structure used as an input for the move generator. This procedure aims at generating all legal moves stemming from the current position in any part of the search tree. Regardless of the type of piece coding which can vary a lot, its common property is the existence of a positional matrix that memorizes the current position found in a part of the search tree. Besides such uncompressed forms of position representation there are various models using compression as well as models which represent positions in readable format, e.g. EPD and FEN representations. Even though they require a lesser amount of memory these formats are practically unusable, especially in endgame positions where one finds many empty squares.

What happens here is that the time needed for compression and/or decompression into/out of such formats is significantly longer than the time spent in mere comparison, which leads us to think that they are not an efficient solution, particularly with regard to the speed of execution of the procedures being processed. On the other hand, we need to solve the problem of the internal representation of the chess board and piece coding in an efficient way too. The representational organization of the chess board [2], depending on the move generation procedure, can be done in different ways, for example as an 8X8, 10X10, 12X10 or 12X12 byte grid. Our program, AXON, has a 12X12 byte matrix structure. We should mention though that it is only to be expected that performance-wise the best results are obtained with an 8X8 grid that is very much like an iconic representation of the chess board.

Implementations of other representational matrices lead to proportional slowdown, but the gains earned by the use of larger grids as compared to the 8X8 grid are faster recognition of the grid borders, one dimensional coding in place of two dimensional one, etc.

In order to explain the core problem better, let us propose a simple type of coding (8X8 bytes). (Let us emphasize first that the form of piece coding is completely irrelevant to our further analysis and serves only as an illustration.) The chess board hosts 12 different pieces (six white and six black pieces), therefore to represent all of them we need four bit coding. The coding can be done in the following way:

**Table 1:** Piece coding

| PIECE | CODE | Dec. |
|---|---|---|
| Empty square | oooo | o |
| White pawn | ooo1 | 1 |
| White knight | oo1o | 2 |
| White bishop | oo11 | 3 |
| White rook | o1oo | 4 |
| White queen | o1o1 | 5 |
| White king | o11o | 6 |
| Black pawn | 1oo1 | 9 |
| Black knight | 1o1o | 1o |
| Black bishop | 1o11 | 11 |
| Black rook | 11oo | 12 |
| Black queen | 11o1 | 13 |
| Black king | 111o | 14 |

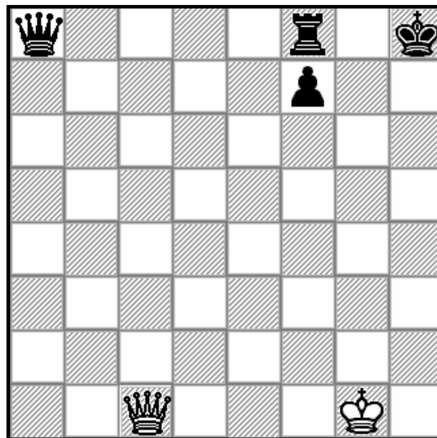

**Figure 1:** White draws by perpetual check.

Figure 1 can serve as an example of a simple position in which White draws by a perpetual check (1. Qh6+). If we use the coding presented in Table 1 the internal representation of the 8X8 matrix would look like this:

| **13** | o | o | o | o | **12** | O | **14** |
|---|---|---|---|---|---|---|---|
| O | o | o | o | o | **9** | O | o |
| O | o | o | o | o | o | O | o |
| O | o | o | o | o | o | O | o |
| O | o | o | o | o | o | O | o |
| O | o | o | o | o | o | O | o |
| O | o | o | o | o | o | O | o |
| O | o | **5** | o | o | o | **6** | o |

**Figure 2:** Coded form of a position (Figure 1).

The positional matrix in its uncompressed format shown in Figure 2 can be used directly as input to an efficient move generator. This matrix is simultaneously part of the position detection procedure. The capacity of the matrix is 64X8-bit, equaling 512 bits, and that position comparison has to be done in the same format making the procedure inefficient. The downside to this approach can be seen rather clearly in the endgame where great portions of the chessboard are empty squares and a great deal of time must be spent on comparisons of relatively similar positions. Another disadvantage is that in long lasting games, with many moves played in the endgame, one has to take into account the length of the "game history" so that the analysis may be extended to tens of other, already played, positions. We will list some of the methods that can help to speed up or eliminate the inflated 512-bit comparison:

In the search stage we start with a position that is currently being evaluated as the deepest in the search tree. The investigation extends towards the root of the search tree. If there is some sort of "relevant history" the search must be extended onto positions stored in the corresponding data structures. If a repeated position is found (such cases are

rare from a statistical point of view) the moves that might otherwise be produced by this node are not generated and the evaluation is not assigned. The value of 0.00 is automatically put down as the evaluation mark instead. Since the comparison is done only in the case when the same side is to move it is necessary to take into account only positions with either even or odd depths depending on whether the current position that serves as a start point is of even or odd ply depth. This helps to reduce the number of possible search queries by 50%.

The search has to be terminated at the moment a position that resulted from an exchange of pieces or pawn advancement was registered. If no position repetition had been registered up to that moment, a conclusion has to be drawn that there was no position repetition whatsoever.

In order to speed up the search a positional checksum (16-bit) can be introduced so that the control checksum should be verified before any comparison of positions. In case the sums do not check out correctly there is no need to investigate complete matrices: they simply have to differ. A part of the hash key variable can be used as checksum, of course if the program has viable hash support.

Using the 64-bit Zobrist key instead of full positional matrix is the probably most popular method for draw detection. The Zobrist keys are the products of the hash based search functions.

Using the 32-bit architecture of modern processors 32-bit registers can also be exploited for comparison purposes. In this manner, we could ensure 16 comparisons at least. With the latest 64 bit processors, we could have 8 comparisons at the most. Machine code can contribute significantly to such speed ups.

Most of the recently available chess engines have procedures developed on the basis of the principles sketched in this short review. It appears to us that it is quite possible to introduce an efficient algorithm which would avoid the inflated 512/64 bit structures, by relying on variant strings of moves coded in the 16 bit format.

### 2.1. Formal definition of the standard solution

At the end of this section, we can define the standard solution in the following concise form:

A finite move sequence of moves is represented as $p_1, p_2, p_3, \ldots, p_n$, where $p_i$ is the position (or 64-bit Zobrist equivalent) after the $(i\text{-}1)^{th}$ move for $i = 2,3,\ldots.n$. Each move between two adjacent positions $p_{i-1}$ , $p_i$ is clearly defined. An assumption is $p_i <> p_j$ for $i <> j$ and $i, j < n$ , which is true when it is applied to game-tree search. The standard method is to check if there is some $i$ such that $p_i = p_n$ for $i = n$-1, $n$-2, …,1. Cost can be halved using the fact $p_i <> p_j$ for all even $i$ and odd $j$ (moves made by different players).

## 3. PROBLEMS ASSOCIATED WITH THE GAME HISTORY AND THE SEARCH TREE

We have mentioned that according to the rules of the game of chess a threefold repetition of a position is considered as a draw. In a real computer game of chess, for a program to have this rule applied successfully, all positions generated throughout the game (beginning from the starting position onwards) have to be memorized in a data

bank. If the Zobrist keys are utilized the 64-bit equivalents are memorized as well and are eventually assigned corresponding flags.

Therefore, it is proposed [4] that all positions be divided into:

1. positions that were played (game history),
2. positions that are generated in the search tree.

If three identical positions have been found in the search tree, the solution is simple: draw should be detected immediately (return score of 0.00). Such cases as the most frequent in practice and, as a rule, occur during testing (most notably in *.epd test suites).

More complex cases of draw detection occur when some positions are in the game history, and others in the search tree:

a) two identical positions are in the game history, and one is also in the search tree,
b) one position is in the game history and two are in the search tree,
c) one position is in the game history and one in the search tree.

Cases under (a) and (b) actually indicate the repetition of an identical position three times so they can be immediately evaluated as a draw.

Cases under (c) can be treated in two ways:

If the search ignores positions that can be subsumed under (c) above then practice shows that this can result in "boring" play, as the same position can be repeated in a game twice before proceeding (in case the program has a better evaluation score than the opponent). Therefore, even if in the cases in which the program assesses its position as superior it wastes time by going back to the same position. The behavior of the program resembles that of some strong master-strength players where the superior side with little time on the clock repeats some moves so as to reach the desired time control.

All positions that can be found under (c) are treated as a draw (AXON uses this approach). This approach enables the program, when it finds itself behind, to attempt to reach a draw by repeating a position, which might lead to some rather awkward looking moves at times.

In chess programming some other methods have also been used, such as those trying to detect only two identical positions be they both in the search tree or with only one in the search tree and the other in the game history.

## 4. NEW APPROACH BASED ON VARIANT STRINGS

The variant strings method that we propose is applicable only to those regular search algorithms that observe the chess rule of alternate move turns (*Alpha-Beta, PVS*, *NegaScout*; *Null move* is excluded here as it permits one side to execute two moves in a turn). For relevant sources referring to the techniques mentioned cf. [1], [2], [3]. These strings are actually move lists that start at the root of the search tree and end up in the terminal node being evaluated. This concept is quite similar to the concept of best-line. As a matter of fact, best-line is a variant that contains best moves for both sides (own side and the opponent). Depending on the manner of coding there might be different types of variant strings. For instance, the main draw string in Figure 1 can be presented as follows:

Qh6+ Kg8 Qg5+ Kh7 Qh5+ Kg7 Qg5+ Kh8 Qh6+ …

Alternatively, we could use a similar, but basically equivalent, positional notation applied in many chess programs (most notably the Winboard GUI) (striving to unify and compress information):

C1H6 H8G8 H6G5 G8H7 G5H5 H7G7 H5G5 G7H8 G5H6 …

Using the positional notation it is easy to obtain information regarding the from- and to- squares (starting and destination piece movements). If we take only a brief glance we can see that 4 bytes (32 bits) are necessary to code one single move. However, it is possible to reduce the number of bytes to 16 in coding one move. Viewing a positional matrix that has 8X8 squares, 6 bits (or 64 combinations) are necessary to code these squares. In the case of a 12X12 matrix 144 squares or 8 bits are required.

*This means that a 16-bit system of coding can be defined in a way that the from- and to- squares are coded by using 8 bits.*

Such a simple coding system has served as a basis for a new procedure for draw detection by help of variant strings. The basic premise is that the information leading to draw detection is not obtained via positions but by help of variants coded in the previous section. In AXON, our chess-playing program, the organization of the move-generator and the tree search system is embedded in the type of coding we have just described in general. Variant strings that are formed in the tree search all the way to the terminal nodes serve as input to the move repetition procedure. The next section will describe and analyze the algorithm we used to achieve this goal and thus speed up the program itself.

### 4.1. New approach and null move heuristic

During the process of draw detection by way of variant strings, the basic assumption is that the side to move changes virtually. While using null move this rule is not in effect as one side is allowed to move twice. In order to detect correctly the branches in which draw detector can be used, the simplest solution is to introduce a *null_move_counter* that will incrementally increase by 1 for each null move procedure inside the main alpha-beta/PVS (negascout) procedure. This implies that at any point in the tree only if the *null_move_counter=0* is it possible to detect the repetition of a position.

### 4.2. Formal definition of the variant string method

The new method could be abstracted in the following form:

The finite move sequence of moves is represented as $(p_1,a_1,b_1)$, $(p_2,a_2,b_2)$,…, $(p_n,a_n,b_n)$ where $p_i$ is the moving man (piece identification) , $a_i$ and $b_i$ are the from- and to- addresses of the *i-th* move on the board, respectively. The algorithm actually enables a repeated transition of moves $(p_i,a_i,b_i)$ and $(p_j,a_j,b_j)$ to be moves $(p_i,a_i,b_j)$ and $(p_i,0,0)$ for $p_i=p_j$ and $b_i=a_j$ and $i<j$. A move $(p,a,b)$ such that $(a=b)$ is called a closed chain. Also symmetric moves $(p_i,a_i,b_i)$ and $(p_j,a_j,b_j)$ where $p_i=p_j$ and $a_i=b_j$ and $b_i=a_j$ generate two closed chains $(p_i,0,0)$ and $(p_j,0,0)$. When no transition is possible, the move repetition

occurs only if all chains are closed and the same side is to move. In both the standard and new methods, a flag indicating whether it is a reversible move or not for each move can be applied to truncate the move sequence and improve the efficiency.

## 5. ALGORITHM

A great majority of contemporary computer chess programs has evolved from some sort of recursion. The basic types of algorithms that have already been mentioned have also been based on recursive procedures. AXON has a recursive kernel which makes the recognition of repetition detection a little more complex as variant strings are formed within the main program stack together with other important parameters crucial to the search (*Alpha-Beta* values, positions, return value addresses, etc.). Since the repetition detection needs access to the variant string starting from the terminal node backwards, this information is not easy to reach through the stack, which is commonplace in recursive programs.

On the other hand, it is quite convenient to use the same procedure together with the game history (a list of previously played moves) so as to maximise the efficiency of the procedure. Having in mind such issues, the authors are willing to suggest the following solution (all data structures and functions are in the standard programming language Pascal).

### 5.1. The definition of the data structure

We will introduce a global definition of the 24-bit data:

```
Move_Type = packed record
              From_square:byte;
                  To_square:byte;
                Piece_Flags:byte;
end;
```

This leads to:

`From_square` : from square,
`To_square` : destination square,
`Piece_Flags` : 4-bit piece code (Table 1) and in continuation 4-bit flag.

Thus, *Piece_Flags:=16*Piece + Flags.*

Together with the piece definition whose move has been registered the information about the color of the side to move is stored so that if the most weighted bit has the value of 1 then it is Black's turn to move, and if that bit is 0 then it is White's turn to move (Table 1). Flags contain bits that mark non-recursive moves: *pawn movement/ capture/ castling/ en-passant*. Thus, if any of the flags is different from 0, a non-recursive move is in question so that the procedure is immediately terminated since no repetition of the position has been found. The same happens if all the chains are closed and the other side is to move just as at the beginning of the analysis. As examples we can use different triangulations in pawn endings. All the information in the mentioned structure of the type *Move_Type* is generated by the *move_generator*.

The basic data structure can be defined as follows:

```
LIST_OF_MOVES: array [0..254] of move_type;     {Move list }
index:integer;                                  {Move list pointer}
CHAIN_LIST: array [1..16] of move_type;         {Concatenation list}
UnClosed_chains: word;                          {The number of open chains}
```

The string *list_of_moves* is supposed to memorise the game history as well as the current string generated as a result of the tree calculation. As a matter of fact, the mentioned string contains 24-bit coded moves starting with the initial move up to the terminal node. Before the initialisation of the iterative search the procedure *Generate_list_of_moves* is invoked and it resets the whole string reading into it 24-bit coded moves taken over from the ancillary string containing the game history. Such a string is being updated during the search so that the procedure enabling the move generator on the basis of the search depth writes the coded moves directly into the string marked as *list_of_moves*. The address of the element that was put down last is written into the variable *move_TOS* of the *integer* type. Defined in this manner, the data represent input parameters that generate the function dubbed *PERPETUAL* whose definition and details will be presented in the next section.

The move coding generated in the procedure *move_generator* is done on the basis of an inherently spatial principle. To speed up the coding, AXON employs a spatial one-dimensional 8-bit address of the from- and to- square. To illustrate this, let us return to the previous section and the first move in the draw line C1H6 (Figure 1). If we code the chessboard unidimensionally so that the A8 square has a value of 0, the B8 square value 1 and the H1 square value of 63 then the move C1-H6 can be coded as 58-23, i.e. instead of two 8-bit numbers it can be replaced by one 16-bit number whose value is 14871 (58*256+23). All other moves can be coded in the same way. If *move_generator* has generated any irreversible move such as a capture, pawn advance or castling this can be marked by setting the bit of the greatest possible weight in the 8-bit code of the from-square. Such an action is possible because we need only 6 bits ($2^6$=64) to code the square itself. All other flags and moving piece are also generated by the *move_generator* and packed into the *Piece_Flags* byte.

The main function *PERPETUAL* aims to process the structure marked as *list_of_moves* and thus return one of the two possible logical values: TRUE if move repetition has been detected and FALSE if not. This procedure hinges on the utilisation of the concatenating *chain_list*.

The algorithm of the *PERPETUAL* function can be generally described using the following steps:

The initial indicator is set to the value *move_TOS* - the address of the last move written in the move list;

All the elements of the concatenating list *(chain_list)* are set to 0;

Scanning the move list starts from the last address *(move_TOS)* towards the lower addresses. In case the list yields an element whose bit is of the greatest weight marked 1 the procedure is abandoned immediately as it is an irreversible move meaning that position repetition has not been detected.

If the move was reversible it is included in the concatenating list, observing the following rules:

If the concatenating list has a move whose to- square is identical with the from-square of the current move then the concatenation is performed by way of the transformation of the moves in the concatenating list. An example: if the concatenating list contains the move A1B1 and the current move is B1E5 then instead of the existing A1B1 the move A1E5 is listed. This rule can be dubbed *the rule of* transition as it is equivalent to the corresponding relation in mathematics. If, after the application of the rule of transition, a complete correspondence between the from- and to- square has been noted the move is removed from the concatenating list (*the rule of* reflexion). An example: if the move A1B1 is in the concatenating list and the current move is B1A1 the application of transition should generate A1A1 so that instead of that the corresponding slot in the concatenating list is filled up by a zero. The basic logic of the procedure is *closing the variant chains. Only in case that all the chains are closed (in other words all elements of the concatenating list are equal to 0) can move repetition be recognised.* Let us illustrate this point by going back to Figure 1. Suppose that the following line is possible: C1H6 H8G8 H6G5 G8H8 G5H6 … Since the scanning is done backwards it is easy to see that after the first two steps two potential chains of moves are possible (G5H6 and G8H8), after the third step the first chain is closed (H6G5), while the fourth step marks the closing of the second chain (H8G8), so that a we can proclaim a case of move repetition. Thus if all variant chains in the concatenating list close at any point then we have move repetition. This procedure is repeated after every move from the move list. If the bottom of the list or any move that has a bit set to the greatest weight is reached the procedure is terminated as move repetition has not been detected. The procedure appears quite efficient and encompasses even far more complicated positions that exhibit multivariant chains

The following figure presents a chess study that can be our testing ground where the efficiency of our move repetition detection method can be assessed:

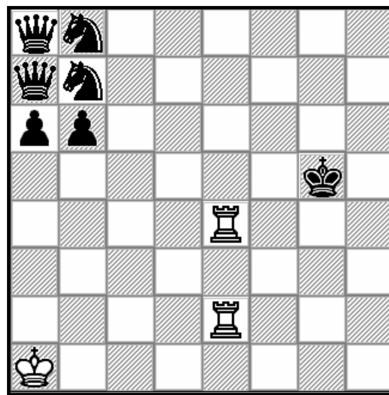

**Figure 3:** Draw by the perpetual check.

It is quite clear that Black has enough potential to free its pieces and to win in just one move. If White is to move it can draw by simply checking the black king, using the rooks along the e-file. It is curious to note that the program needs some time and a very deep calculation to realise that, regardless of the complex routes that the black king

can take along the three free verticals, the position should be evaluated as 0.00. Such positions are very good at showing how advantageous the new principle of move repetition detection can be one. The new principle is based on highly efficient 24-bit variant chains that could contain dozens of elements but are still less costly than a great number of very similar 64-bit coded positions.

The next example (Figure 4), is a well-known position from the frequently quoted Bednorz-Toenissen test suite (BT2630, No.7). Black, after playing 1…Rd6! forces White's 2.Qxd6 and after …Qe3+ gains a perpetual check. This example can serve as a test for our draw detection procedure within *Qsearch* (as the majority of forced checks, with the white king being able to move along e1, f1, g1, g2 and h2 squares, is processed in *Qsearch*). AXON solves this particular problem at ply 9. Other programs, which go much deeper in order to detect checks in *Qsearch* (e.g. DEEP FRITZ) find the solution at a lesser depth (ply 6-8).

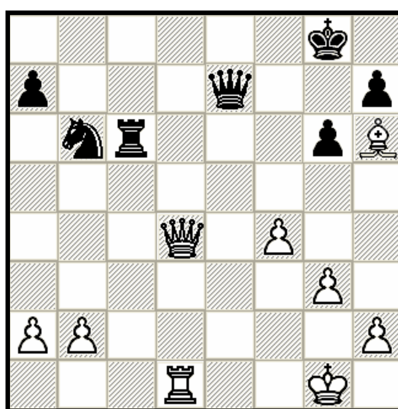

**Figure 4:** Black forces a draw (1…Rd6!).

### 5.2. Formal definitions of the rules

The rules for the manipulation of the existing data structures could be formally defined as follows:

**The rule of transition:** If the current move has the form of *Piece_AnBm* and in the concatenation string *chain_list* there is a move of the form *Piece_BmCq*, the move in the concatenation *chain_list* is replaced by *Piece_AnCq*.

**The rule of reflection:** If the *chain_list*, after the application of the rule of transition, contains the move that has the form of *AnAn*, this move is deleted from the list. The counter *unclosed_chains* is decreased by 1.

**The rule of symmetry:** If, after the application of the rule of transition in the string *chain_list* move of the form *Piece_AnBm* and *Piece_BmAn* can be found, they are deleted from the list. The counter *unclosed_chains* is decreased by 2. The rule of symmetry detects cases when two identical pieces switch their positions.

### 5.3. The identical piece place-switching issue

In the majority of cases the rules of reflection and transition are sufficient to register the repetition of a position correctly. However, there are some cases of identical pieces switching places which are successfully treated by help of the rule of symmetry. The following diagram represents the starting position using two knights only:

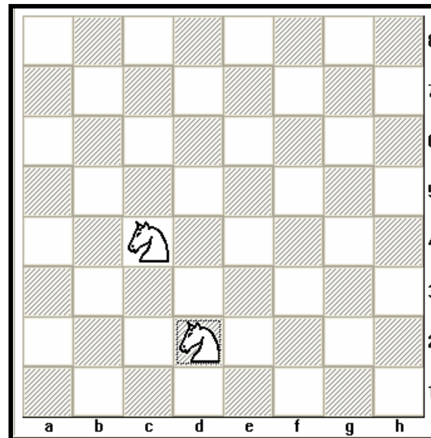


**Figure 5:** Illustration for the place-switching problem.

Let us presume that the following ensued:

C4A3 D2C4 A3B1 B1D2

As a result an identical position as the starting position has been created, with the knights at c4 and d2 switching places. The following table can illustrate the situation with typical move-after-move variables (the second column also has the information about the pieces):

**Table 2:** The solution of the place-switching problem

| Move | LIST_OF_MOVES | UnClosed_chains | Rule applied |
|---|---|---|---|
| C4A3 | N_C4A3 | 1 | Insert |
| D2C4 | N_C4A3 N_D2C4 | 2 | Insert |
| A3B1 | N_C4B1 N_D2C4 | 2 | Transition (c4a3 and a3b1) |
| B1D2 | N_C4D2 N_D2C4 | 2 | Transition (c4b1 and b1d2). **Rule of symmetry applied** (N_c4d2 and N _d2c4). |
| --- | --- | 0 | Repetition detected. |

The application of *all three rules* defines a complete algorithm for draw detection by position repetition by help of variant strings, applicable to _all_ cases that may arise.

### 5.4. The definition of the function *PERPETUAL*

This subsection describes the definition of the algorithm that we discussed in the previous section. The equivalent procedure in assembly is part of the chess program AXON whose kernel is completely machine coded. The function PERPETUAL that processes the structure *list_of_moves* at the input has a variable *move_TOS* which is the last value in the list. As output it has TRUE if a repetition has been found or FALSE if that is not the case.

In brief, the algorithm's functioning can be described as follows:

The algorithm starts from the last move in the variant string going backwards.

If some non-reversible move or the beginning of the list is found the procedure is terminated because no repetition was recognized.

For each move a scanning of the concatenating list is performed *(chain_list).*

If in the concatenating list a move is found that the rule of transition can be applied to the rule is executed, after which a probe is done for the program to check if the rules of reflection or symmetry can be executed as well (the *unClosed_chain* variable is decreased by 1 or accordingly in case of the execution of these rules).

In case that the concatenating list does not contain a move that the rule of transition can be applied to, the list is extended with a new element (current move) and the variable *unClosed_chains* is increased by 1.

If, at any moment, the value of the variable *unClosed_chains* is 0, the procedure is terminated since the repetition of a position has been detected.

The procedure which definition we have just described is efficient in its execution and appears to be as fast as the already existing procedures of move repetition and draw detection due to the advanced algorithm that was implemented in AXON. In practical play the significance of this procedure is even greater especially in endgames with rooks or queens and exposed kings where checking sequences can be very long. Significant portions of the search tree in which move repetition is processed are not considered but are simply replaced by the zero value. Thus enhancing the tree search speed indirectly. [1]

## 6. A MATEMATICAL FRAMEWORK OF THE EFFICIENCY COMPARISON OF THE STANDARD AND NEW APPROACHES TO DRAW DETECTION

The comparison of efficiency of these two methods is quite complicated due to a high number of parameters which may influence the processing speed. Nevertheless, we shall try to formalize a framework for their comparison:

[1] See Appendix for an illustrative game that AXON played against a program estimated to be >2650 ELO at the sudden death level of game in 10 minutes (Diep-Axon).

### 6.1. The standard method and the implementation of 64-bit Zobrist keys

If the sequence of **n** moves is given after which there was a repetition and if the time $T_z$ is the time needed for the transformation of 64-bit keys between positions, then the aggregate time needed for the standard method would be:

$\mathbf{Z=64nT_z}$ (64-bit for the comparison and transformation)

### 6.2. The variant strings method

If the sequence of **n** moves is given after which there was a repetition and if between these two sequences *k* pieces moved (in practice k is a small figure, usually 2 to 4), and if $T_v$ is the time devoted to the processing of the linear list of moves, the function should look like the following:

$\mathbf{V=24knT_v}$ (24-bit for the comparison)

The critical parameters in both functions indicate a high level of dependence on the "fuzzy" parameters (those that cannot be exactly determined), such as the type of the position in the tree, current depth, ordering of pieces, as well as the processor-type and compiler types (32 or 64-bit), etc.

As a general conclusion that appears to be confirmed by actual play endgames with only a few pieces such as $k$, $T_v >> 0$, the variant strings algorithm indicates high efficiency. As opposed to our algorithm, whose time function is exceptionally non-linear, the standard method has a comparatively stable time function.

## 7. EXPERIMENTAL DATA

In order to illustrate the influence of the described procedure on the execution of the AXON chess program we will use a chess position from the *wac.epd* chess suite that is widely spread among computer chess programmers. It is a rook ending with both kings exposed (Figure 3). Similar positions appear rather frequently in over-the-board tournament chess. The version of AXON we used was slightly modified with a null move generator implemented as well as some other pruning techniques. On the other hand we turned off the heuristic knowledge of endgames so as to make the results dependent almost solely on the search algorithm. Endgame tablebases (such as the Nalimov tablebases) were not used. The hardware platform was a PC running on an AMD Athlon 2200+ processor with 256 Mb RAM. The test was actually an analysis of the position with the algorithm for move repetition turned on and off. The search depth was iteratively increased from ply 1 to ply 12 in both modalities (move repetition algorithm turned on and off) and the recorded parameters were: best key move at a given depth, evaluation, node count, total count of terminal and generated positions in both modalities, as well as the relative speed of generation and the length of the time spent in search.

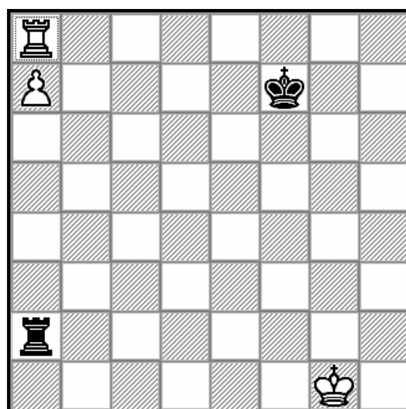

**Figure 6:** White to move.

The solution leading to the winning sequence is naturally Rh8!, thus making it possible for the white king to make a smooth journey to the solitary white pawn which will soon be promoted to a queen. Meantime the black rook can check the exposed white king for quite some time. The following tables present the data obtained from the program:

**Table 3:** The analysis of the position **without** the move repetition algorithm on

| Depth (ply) | Key move | Evaluation | Node count |
|---|---|---|---|
| 1 | G1F1 | +2.30 | 929 |
| 2 | G1F1 | +2.23 | 1396 |
| 3 | G1F1 | +2.44 | 7299 |
| 4 | G1F1 | +2.37 | 18964 |
| 5 | G1F1 | +2.42 | 43701 |
| 6 | A8H8 | +2.47 | 274992 |
| 7 | G1F1 | +2.48 | 364926 |
| 8 | G1F1 | +2.52 | 1541540 |
| 9 | A8H8 | +2.55 | 2252388 |
| 10 | A8H8 | +2.55 | 2062293 |
| 11 | A8H8 | +2.65 | 4635873 |
| 12 | A8H8 | +5.04 | 9770293 |

The total count was: 1 193 867 terminal nodes evaluated and 20 974 594 positions generated.

**Table 4:** The analysis of the position **with** the move repetition algorithm turned on

| Depth (ply) | Key move | Evaluation | Node count |
|---|---|---|---|
| 1 | G1F1 | +2.30 | 929 |
| 2 | G1F1 | +2.23 | 1396 |
| 3 | G1F1 | +2.44 | 7299 |
| 4 | G1F1 | +2.37 | 18883 |
| 5 | G1F1 | +2.41 | 46990 |
| 6 | A8H8 | +2.47 | 263862 |
| 7 | G1F1 | +2.48 | 399843 |
| 8 | A8H8 | +2.47 | 2806126 |
| 9 | A8H8 | +2.55 | 693285 |
| 10 | A8H8 | +2.55 | 1451126 |
| 11 | A8H8 | +2.55 | 3618668 |
| 12 | A8H8 | +4.87 | 8524217 |

In this case the total count was: 977942 terminal nodes evaluated, 17 832 624 positions generated. 14 941 cases of position repetitions were noted.

The conclusions to be drawn from the data are as follows:

The structure of the key moves and the evaluation in both experiments were similar, the differences were generated because of the activity of the extending search algorithm in AXON that adapts to the search depth (adaptive depth control).

The comparative increase of the positions searched is similar in both cases. However, the increase of the search depth using the move repetition algorithm is evidently beneficial here. This is so because the repeated position is detected at a shallower depth and such positions are evaluated as 0.00 so that the sub-trees stemming from them are pruned and not extended any further as is the case in the first part of the experiment. The second part of the experiment shows that the algorithm is slower by about 10%, the time taken up by the move repetition procedure. The decrease in speed is smaller in the middle game or in positions where there are many non-reversible moves.

The tree being generated in the second part of the experiment is smaller by 18% judging by the number of terminal positions, and by 15% counting the number of generated positions. The analysis performed in the second modality has a shorter duration by 10% than the one in the first modality with the move repetition algorithm off.

The conclusion to be drawn from our experiment is that the move repetition detection procedure that is based on the method utilised by the authors processes fewer positions and creates a smaller number of terminal nodes thus compensating for the loss of general processing speed. The procedure is beneficial in terms of overall performance achieving a gain of approximately 10%. In endgames with queens and exposed kings this percentage should prove greater. Hopefully, these are credible and convincing experimental results justifying the inclusion of the discussed move repetition detection algorithm into state-of-the-art computer chess programming.

## 8. CONCLUSION

The authors have proposed a novel approach to the problem of move repetition detection using variant strings as opposed to the matrix-based approach (with Zobrist keys) used in the majority of the current chess playing programs. The primary aim of the authors has been a solution to the problem of draw detection by position repetition in cases where the standard method of utilizing the Zobrist keys is not optimal, i.e. when the Zobrist keys are not generated in the tree or no hash memory is used. Such cases are frequent in the implementation of fast *Qsearch* routines. The procedure in the paper, on the contrary, hinges on another type of data (variant strings) that is present in every search as a by-product of the move generator.

As already stated, the efficiency of the two approaches is difficult to assess since the new method is extremely non-linear with high speedups in the endgames with only a few pieces. The authors suggest a complementary implementation of the two approaches, meaning that in the systems which are based on hashing and the Zobrist keys the standard 64-bit positional equivalents ought to be used, whereas when such an approach is not suitable the variant strings method is recommended. Hopefully, this complementarity should cover all cases and draw detection can be performed throughout the search tree regardless of hashing. The authors have attempted to prove the efficiency of their treatment of the problem in both the areas of implementation and the speed of execution. The algorithm proposed by the authors has been discussed in detail. For the sake of transparency, this algorithm is described and presented in Pascal whose machine equivalent, opaquely coded in assembly, is a constituent part of AXON. It was also used in a small experiment relating to the discussed issue of move repetition detection by means of which the authors contend that the implementation of their procedure leads to considerable gain (10% overall) in the processing of the search tree, especially in endgames. The current version of AXON does not utilize the Zobrist keys as it uses a non-standard type of hashing. The authors are planning to implement the standard method in the new 64-bit version of the program. Hopefully, this will make possible a much more precise comparison of the efficiency of both methods.

Generally speaking, the new method can be used in other 2-player matrix games (naturally, perfect information is required) where it is possible to code the game lines by help of variant strings. Examples could be Chinese and Japanese chess (Xiangqi and Shogi). It goes without saying that the benefits eventually arising from the draw detection algorithm depend on the nature of the rules of the game in question (since in the said games the repetition of a position does not imply a draw as there is no perpetual check).

**Acknowledgements:** The authors would like to express their gratitude to the anonymous referees for their useful and constructive advice. The deficiencies of the paper are, needless to say, only those produced by the authors themselves.

## REFERENCES


[1] Donninger, C. "Null move and deep search: selective-search heuristics for obtuse chess programs", *ICCA Journal,* 16 (3) (1993) 137-143.

[2] Fray, P.W., *A introduction to Computer Chess. Chess Skill in Man and Machine (Texts and monographs in computer science)*, Springer-Verlag, New York, N.Y., ISBN 0-387-07957-2, 1977, 55-67.

[3] Kaindl, H., Horacek, H., and Wagner, M. "Selective search versus brute force", *ICCA Journal,* 9(3) (1986) 140-145.

[4] Moreland, B., Zobrist keys description could be find at: http://www.brucemo.com/compchess/programming/zobrist.htm, Move repetition discussion at: http://www.brucemo.com/compchess/programming/repetition.htm, 2001.

[5] Zipproth, S., "Suchet, so werdet ihr finden, Eine Reise in die Gedankenwelt eines Schachprogramm", *Computer schach und spiele*, 3 (2003) 15-19.

[6] Zobrist, A.L., "A new hashing method with applications for game playing", *ICCA Journal*, 13(2) (1990) 69-73.

[7] Vučković, V., "Realization of the chess mate solver application", *Yugoslav Journal of Operations Research (YUJOR)*, 14(2) (2004) 273- 288.

[8] Vučković, V., "Decision trees and search strategies in chess problem solving applications", *Proceedings of a Workshop on Computational Intelligence Theory and Applications*, SCG, Niš, 141-159, 2001.


## APPENDIX

[White "DIEP version=2.00"]
[Black "AXON"][Result "1/2-1/2"]

1. d4 Nf6 2. Bf4 d5 3. Nd2 Nc6 4. e3 Bf5 5. Bb5 a6 6. Bxc6+ bxc6 7. Ne2 Rb8 8. b3 e6 9. O-O Bb4 10. Nf3 O-O 11. Ne5 Rb6 12. c4 dxc4 13. bxc4 Bd6 14. c5 Bxe5 15. cxb6 Bxf4 16. Nxf4 cxb6 17. Rc1 Qd7 18. Qa4 g5 19. Ne2 Bd3 20. Rfe1 c5 21. Qb3 Bxe2 22. Rxe2 cxd4 23. Rd1 Nd5 24. Rxd4 Rc8 25. Rc2 f5 26. Rxc8+ Qxc8 27. Rc4 Qd7 28. Qc2 Qg7 29. Rc8+ Kf7 30. Qd1 Kg6 31. g4 Qf6 32. Rg8+ Kf7 33. Ra8 fxg4 34. Ra7+ Kg8 35. Qa4 Kf8 36. Qxa6 Qa1+ 37. Kg2 Qc1 38. Qb7 Nxe3+ ! 39. fxe3 Qd2+ 40. Kg3 Qxe3+ 41. Kxg4 Qf4+ 42. Kh3 Qe3+ 43. Kg2 Qd2+ 44. Kf3 Qd3+ 45. Kf2 Qd2+ 46. Kg3 Qf4+ 47. Kg2 Qd2+ 48. Kf3 Qd3+ 49. Kg2 Qd2+ ½ : ½

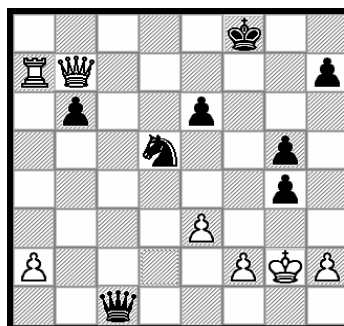

**Figure 7:** Black to move. 38….Nxe3+ ! draw

AXON plays 38… Nxe3, after seeing that White wins by mating in the next move. However, the checking sequence is long and needs to be extended to ply$^n$ that is quite difficult to calculate using the traditional approach to the move repetition problem. By drawing on the described procedure of move repetition detection, AXON was able to extend the search and draw the game.