A fast software-based method for upcoming cycle detection in search trees

ir. M. N. J. van Kervinck (M.Sc.)
marcelk@bitpit.net

April 6, 2013 - for preview

Abstract
An algorithm is presented that detects cycles one ply before they appear in the search of a game tree. The algorithm is suitable for use in the leaf nodes of a chess program and can there shift the detection of repetition draws to an earlier iteration. The algorithm is fast because it doesn't generate or iterate over candidate moves during search. Instead, the Zobrist hashes of potential reversible moves are precalculated and stored in a cuckoo table. Further speed is gained by employing a light-weight measure for the displacement of opponent pieces that is based on the exclusive *OR* of an even number of hash history items. Measurements are given for tree size, node speed and improved playing strength when applied to a chess program.

1. Introduction

When in chess one observes the correlation between the evaluations returned by short searches versus long searches an anomaly from the expected cloud around *x=y* becomes apparent: there is distinct set of points on *x=0* and even more on *y=0*, together forming a cross along the axes. See figure 1.

The presence of this cross stems from draws by repetition. It is clear that these draws make up a tactical category that is particularly poorly anticipated by a heuristic search until in the search tree a cycle indeed gets closed. The score disruption suggests that an early detection of such cycles should be beneficial to a chess program.

However, it is not obvious how to do that in software without slowing down the search too much. The chips of Deep Blue[1] contained a hardware repetition detector that could sense upcoming repetitions efficiently thanks to the parallel operation of the comparisons in its 32 ply circular buffer. As Hsu noted: “Also, normal software repetition detectors cannot tell us that a position is about to repeat.”

This paper describes the derivation of a fast upcoming repetition detector in software.

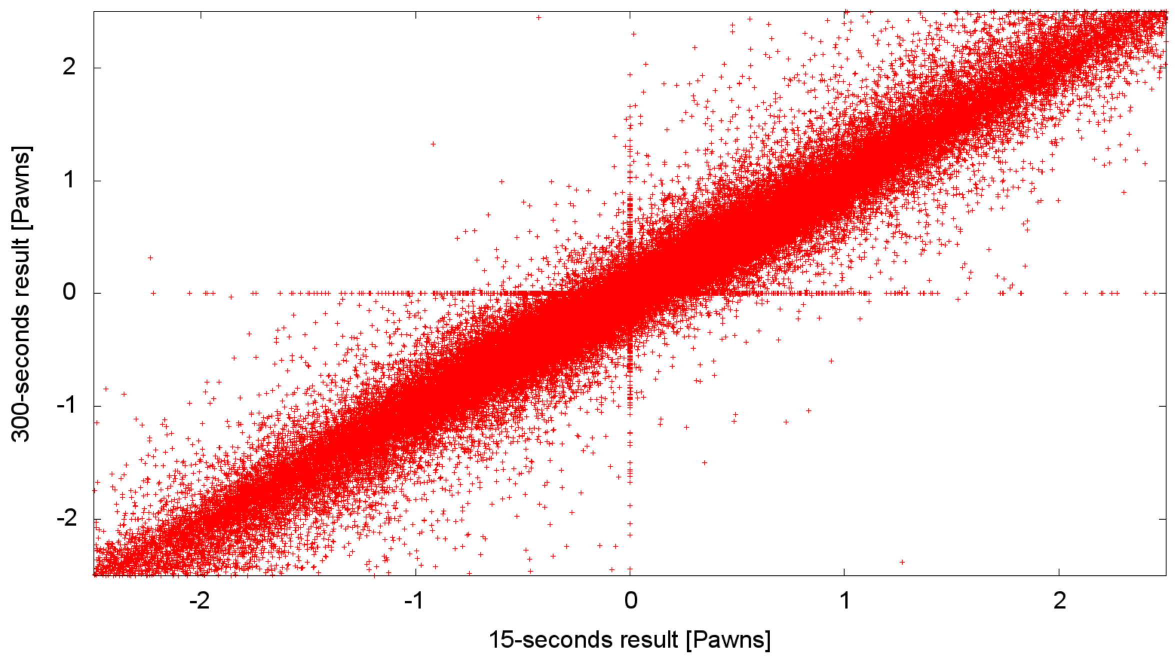

Figure 1. Correlation between short and long searches (100,000 positions)

2. Regular repetition detection

Figure 2 recapitulates the basic algorithm for repetition detection as found in many chess programs. It presumes a stack `S[]` of previous Zobrist position hashes. `S[0]` represents the hash of the current position in the search. `S[d]` is the hash *d* ply towards the root. The number of irreversible moves made before the current position is commonly called the halfmove clock and is represented by the `hm` parameter. The halfmove clock serves both in the early return condition and limits how far up the stack one must look for a potential match. For the sake of simplicity it is assumed that `hm < len(S)`. In other words: the stack may lead back to positions before the root of the search and into to the game history when necessary. If the search employs the null move heuristic[2] such move will be considered an irreversible move that resets `hm` to zero when it is made on the internal board.

```
bool test_repetition(int hm, zobrist_t S[])
{
  if (hm < 4) return false
  for (int d=4; d<=hm; d+=2) {
    zobrist_t diff = S[0] ^ S[d]
    if (diff == 0) return true
  }
  return false
}
```

Figure 2. The basic algorithm for repetition detection

This algorithm is efficient for the branch prediction friendly `hm < 4` condition and the observation that in general search trees contain many captures and therefore `hm` is usually a low number. As the basic algorithm is operating on hashes and not on the positions themselves there is always a remote possibility that a hash collision causes a wrong answer. In a heuristic based search the impact of that can be safely ignored.

For completeness figure 3 gives the supporting code to the basic algorithm.

```
typedef zobrist_t unsigned long long
zobrist_t Zobrist[12][64]

// Pseudo-random number generator for 64-bit numbers
void seed_random64(unsigned long long seed) // No implementation here
unsigned long long random64(void) // No implementation here

void init_zobrist()
{
  // Create Zobrist hash constants
  seed_random64(0)
  for (int piece=1; piece<=12; piece++)
    for (int square=0; square<64; square++)
      Zobrist[piece-1][square] = random64()
}

zobrist_t hash_position(int board[64], int side_to_move)
{
  // For simplicity, the castling and en-passant statuses are ignored here
  zobrist_t hash = 0
  for (int square=0; square<64; square++)
    if (board[square] > 0) hash ^= Zobrist[board[square]-1][square]
  return (side_to_move == 0) ? hash : ~hash
}
```

Figure 3. Basic definitions

The values of `S[]` have presumably all been calculated by the `hash_position()` function or, more likely, were obtained by an incremental update to match the function’s result. This is all completely standard except for the many different methods to incorporate a side to move indicator into the hash value. For the discussion in this paper the hashes for positions with Black to move are *XOR*-ed with an all-ones value, implemented by the logical *NOT* operator ('~').

Going back to `test_repetition()`, the reader will have noticed that the matching test is normally expressed directly as

```
if (S[0] == S[d]) return true
```

instead of going through the temporary variable `diff` and the *XOR* operation in

```
zobrist_t diff = S[0] ^ S[d]
if (diff == 0) return true.
```

Compilers can be expected to emit the same quality of code for both forms. The `diff` form has the advantage that it leads to explore what is necessary to detect a cycle one ply before it actually appears: In that case `diff` will represent the *XOR*-wise Zobrist hash difference between the two positions before and after the final move that closes such cycle. For a reversible move the identity

```
diff == ~(Zobrist[piece][from] ^ Zobrist[piece][to])
```

must be observed as well as

```
diff == S[0] ^ S[d].
```

We call this value the Zobrist hash of the move *(piece, from, to)*.

3. Towards upcoming repetition detection

3.1. First adaptation

With the observation from the previous section one can transform the basic algorithm into one that detects upcoming repetitions as is shown in figure 4.

```
bool test_upcoming_repetition_prototype_1(board, int hm, zobrist_t S[])
{
  if (hm < 3) return false
  for (int d=3; d<=hm; d+=2) {
    zobrist_t diff = S[0] ^ S[d]
    if (is_hash_of_legal_move(diff, board)) return true
  }
  return false
}
```

Figure 4. The basic algorithm adapted

The first change is to iterate over the odd-indexed stack items starting from 3. The second change is the assumption of some method that efficiently determines if `diff` indeed represents the hash of a legal move in the current position. For that some additional knowledge is needed about the current position that is not available in the hash stack. This is taken care of by the `board` parameter. The specifics of this parameter don't need to be further detailed at this point.

It must be noted that the choice for this `board` parameter ignores an alternative line of thinking that is worth briefly mentioning: Instead of considering move legality from the current position one can also consider legality from the `d` ply up positions, because for a cycle to form the reversed move must be a legal move from an earlier position as well. This can be tested by making sure that for these `d` ply up positions all reversible moves have been generated. As there are far fewer `d ≥ 3` nodes than that there are leaf nodes, and each list needs to be generated at most once but can be referenced many times down the tree, the generation cost will not have that much of an impact as compared to performing move generation in the leaf nodes. More importantly, in the nodes of interest presumably a complete move generation has been done already because another reversible move is already being searched from there and many move generators produce all such moves at once. This concept is still somewhat problematic because it needs an efficient method to locate `diff` in all these move lists. Possibly a counting Bloom filter can achieve this. Otherwise either by linear search, or by sorting the lists and performing a binary search. None of these seems very attractive compared to the solution presented below.

3.2. Presence of a reverting move

The next step in the transformation is to find an implementation for `is_hash_of_legal_move()`. The trick is to pregenerate all potential reversible moves on an empty board and calculate their Zobrist hashes. In principle, with a statically defined move set the programmer should have more methods available to efficiently search for an element in that set as compared to when dealing with dynamically generated move sets. One such method is to generate at compile time a gigantic `switch`-statement and let the compiler figure it all out. From current day compilers one can expect at best a binary decision tree with very poor branch prediction however. Another method is to create a hash table indexed by part of the bits in the move hash. The problem of a regular hash table is that it either needs a lot of space to prevent collisions, or it needs an arbitrary number of probes. A perfect hashing scheme with a small footprint seems unobtainable.

The solution comes in the form of the cuckoo hash table technique[3]. Cuckoo tables are hash tables that guarantee that an item can be found in a predetermined number of probes, normally two. In that case the space overhead of the table is about a factor of two. The cuckoo table uses two hash functions on the item to be stored (which is a hash itself here). We call them `H1()` and `H2()`. The next prototype is given in figure 4 and the supporting definitions for the cuckoo tables are in the appendix. The precalculated support arrays `A[]` and `B[]` hold the squares between which the found move goes and can be used to determine its legality in the actual position as the table `Cuckoo[]` only holds further incomprehensible Zobrist hashes of the candidate moves. In chess there are 7,336 reversible move hashes or half that number if we equal *(piece, a, b)* to *(piece, b, a)* as is the case with *XOR*-based Zobrist hashing. The size of the cuckoo table is therefore taken as 8,192 elements and the functions `H1` and `H2` can simply each select 13 different bits from the 64-bit move hash.

```
bool test_upcoming_repetition_prototype_2(board, int hm, zobrist_t S[])
{
  if (hm < 3) return false
  for (int d=3; d<=hm; d+=2) {
    int i = H1(diff)
    if (Cuckoo[i]==diff || (i=H2(diff), Cuckoo[i]==diff))
      if (is_legal_move(board, A[i], B[i]))
        return true
  }
  return false
}
```

Figure 4. The algorithm with is_hash_of_legal_move implemented

3.3. Legality of a reverting move

After it has been established that `diff` represents a valid move on an empty board one must check that it is also legal on the actual board. There are four conditions that could make the candidate move illegal:

1. The move belongs to the other side
2. The path is blocked by another piece
3. The move leaves the own King in check
4. The hash, which represents the two moves *(piece, a, b)* and *(piece, b, a)*, but neither of which is a legal move due to a hashing collision with a combination of other moves

Condition 1 is realistic in prototype 2 and in fact makes it fail as it is. One solution is making two cuckoo tables, half in size each: one with White's move hashes and one with Black's. It means that the algorithm must be aware of the side to move, which is not a big change. However it turns out that the speedup discussed in the next subsection also cures this condition.

Condition 2 can't be avoided. Prior to declaring a match it must be checked that in case of a sliding move all of the squares along the path are free. Even if the opponent pieces have all reverted, it is still possible that the prospective move crosses a piece of either side. This happens easily, for example with a Queen having made two moves along a triangle but finding it impossible to move back to the original square along the third edge because that is blocked. There seems no better way to verify this legality than to examine the vacancy of the squares along its path.

Condition 3 can be ignored because one can infer from the position appearing at level `d` up in the tree that the resulting position is legal.

Condition 4 was tested for empirically and didn't appear in deep searches from 50,000 randomly selected positions. This can be expected as it relates to the strength of the underlying hashing scheme.

Given these considerations and ignoring case 1 for the moment, one can therefore implement `is_legal_move()` with `path_is_clear()`. See figure 5. In bitboard-based programs this function can be based on the occupancy bitboard, a precalculated table and a mask operation. In mailbox-based programs a loop is required. It is noted that once this test must be performed it is almost always succeeding and therefore the overhead doesn't matter much.

```
bool test_upcoming_repetition_prototype_3(board[64], int hm, zobrist_t S[])
{
  if (hm < 3) return false
  for (int d=3; d<=hm; d+=2) {
    int i = H1(diff)
    if (Cuckoo[i]==diff || (i=H2(diff), Cuckoo[i]==diff))
      if (path_is_clear(board, A[i], B[i]))
        return true
  }
  return false
}
```

Figure 5. The algorithm with is_legal_move implemented

3.4. Displacement of opponent pieces

The cuckoo table solution still leaves some to be desired because there are two probes at each level in the stack. The branch prediction is not a problem because both probes are expected to fail most of the time. However, the probes require two memory accesses, and even though the cuckoo table is expected to reside completely in the L2 cache this is still a lot of effort for in the leafs of a chess engine. An indicator is desired that helps avoid performing these probes in the first place. The best such indicator comes from the problem domain: for a repetition move to be present, the other side's pieces must already have reverted to their original location. It would be nice if there is a cheap way to track opponent piece displacement.

The first thought is to dedicate several bits in the Zobrist hash to each side: for example, 10 bit positions are blanked out in the Zobrist table for all White pieces, and 10 other bits are blanked out for all Black pieces. The other side's pieces have likely reverted if the side to move's group of bits in `diff` are all zero. The problem is that this weakens the strength of the Zobrist hashes in general and that it still generates a considerable amount of false positives. Another idea is to track the from and to squares of the moves along the path up the stack and base a displacement measure based on that, for example by summing `(to - from) << piece`. Wherever this checksum yields zero the pieces likely have reverted to their original squares. For this the sequence of moves examined must be available.

Working from this last concept a more elegant method can be found using the information already present in the hash stack `S[]`. Observe that the last opponent's move before reaching the current position has a move hash of `S[0] ^ S[1]`. This expression includes the change of the side to move status which must be excluded when considering the arrangement of pieces alone. Without the side to move status the displacement can be thus be expressed as `~(S[0] ^ S[1])`. Then observe that the opponent's last but one move displaces its pieces furthermore by `~(S[2] ^ S[3])`. Going further up each opponent's move contributes `~(S[2*n] ^ S[2*n+1])` to the total displacement. Therefore in order for the opponent's pieces to revert to their original positions, this sequence of move hashes must cancel out to zero under *XOR*. This method is faster and about twice as precise as the earlier proposed checksumming method.

3.5. The completed algorithm

This is the observation that leads to a rather tight guard before the cuckoo table probes. The complete algorithm becomes figure 6.

```
bool test_upcoming_repetition(board[64], int hm, zobrist_t S[])
{
  if (hm < 3) return false // Enough reversible moves played
  zobrist_t other = ~(S[0] ^ S[1])
  for (int d=3; d<=hm; d+=2) {
    other ^= ~(S[d-1] ^ S[d])
    if (other != 0) continue // Opponent pieces must have reverted
    zobrist_t diff = S[0] ^ S[d]
    int i = H1(diff)
    if (Cuckoo[i]==diff || (i=H2(diff), Cuckoo[i]==diff)) // ‘diff’ is a single move
      if (path_is_clear(board, A[i], B[i])) // Move is legal here (no obstruction)
        return true
  }
  return false
}
```

Figure 6. The completed algorithm for upcoming repetition detection

In the majority of iterations nothing but the stack is probed, as is in the original algorithm. The accumulated *XOR* of an even number of hash history items, modulo the side to move indicator, can be kept in a CPU register and is therefore very fast. The probes to the cuckoo table are avoided until the displacement condition is satisfied. Also the confusion of case 1 in the previous subsection doesn't occur anymore as there can be no opponent move matching `S[0] ^ S[d]` when its pieces in both positions are at the same location. Both White's and Black's move hashes can be therefore be stored in the same cuckoo table.

3.6. Use in scout

Different from the regular repetition detection, which immediately yields a definite score and stops deeper searching, the upcoming repetition merely gives a lower bound as it signals the presence of a single move that yields a draw score and says nothing about the other moves. It is therefore generally only useful in nodes where a draw score raises alpha.

In scout, which are nodes where `alpha+1 == beta` and representing the vast majority of the nodes in a PVS-based heuristic search, it is possible therefore to avoid the function call by first testing `alpha` against the draw score.

```
if ((alpha < 0) && test_upcoming_repetition(board, hm, S))
  return 0
```

It is questionable if this is optimal as the test on `alpha` is hard on the branch predictor. Better results can be expected from taking the first condition out of the function:

```
if ((hm >= 3) && (alpha < 0) && test_upcoming_repetition(board, hm, S))
  return 0
```

3.7. Use in fail-soft alpha beta

In case of a fail-soft framework one must be a bit careful in leaf nodes. When `alpha` is non-negative, even though an affirmative result from `test_upcoming_repetition()` will not raise `alpha`, it can nonetheless impact the return score: whenever the evaluation and quiescence search both come back negative while an upcoming repetition is present, still a draw score should be returned. Not doing so would cause a too-low upper bound to be returned from the search, potentially leading an unneeded inconsistency during research. This subtle and arguably minor concern is not present in a fail-hard framework.

4. Further optimizations

Space can be saved by being more economical with the `Cuckoo[]` table, `A[]` and `B[]`. One can safely store just the lower 32 bits of `diff` in `Cuckoo[]` and store bytes in `A[]` and `B[]`, yielding:

```
        unsigned long Cuckoo[0x2000]
        char A[0x2000], B[0x2000]
        #define K(h) ((unsigned long)((h) & 0xffffffff))
…
        if (Cuckoo[i]==K(diff) || (i=H2(diff), Cuckoo[i]==K(diff))) … etc … .
```

With this the memory for the tables is reduced to 48 kB. This can be further improved by using three hashes instead of two to compact the cuckoo table further.

5. Performance

5.1. Tree size and node speed

The algorithm has been implemented in the author’s chess program Rookie 3.7. Rookie is a mailbox based program and therefore the function `path_is_clear()` loops over the board. To measure the effect two variants are studied, the only difference being the presence of `test_upcoming_repetition()`. Mind that the regular check for repetition is untouched as the two algorithms exist next to each other. Each variant is compiled with the gcc 4.7 compiler using performance guided optimization (PGO). From a database of Grandmaster games 1,000 positions were randomly selected and each searched to 16 ply on a laptop with a 2.6~3.6GHz

Intel i7-3720QM processor with 4 physical cores (8 virtual) and 8GB of memory. The transposition table was set to 256MB and cleared for each position. Just one core was used and the other cores kept idle. The higher clock rate is applicable during the experiment. Node counts and processing times are listed in figure 7.

	Nodes searched		**Search time (average)**	**Node speed**
	Arithmetic mean	Geometric mean	[s]	[Mnps]
***Without* upcoming cycle detection**	61,737,840	29,025,050	26.900	2.489
***With* upcoming cycle detection**	61,954,608	28,969,405	26.969	2.490
Delta	*+0.35%*	*-0.19%*	*+0.3%*	*+0.0%*

Figure 7. Search statistics

Node cycle time is hard to measure precisely on current day computers due to variations introduced by the systems themselves. For that an additional measurement is done on a selected position with a deep search repeated 2,000 times. From this it is obtained that the speed penalty incurred by the algorithm is 5 to 6 clock cycles per node. Not bad considering it is effectively using that time to identify a legal chess move leading to an arbitrary earlier position.

5.2. Strength

To measure the strength difference a match of 12,000 ultra-fast games was played between both programs. The match was run on an AMD Phenom X6 1090T 3.2 GHz with 6 cores and 8GB of memory. The openings were forced from a book of 6,000 starting lines, each played once from both sides. Learning and pondering were disabled and the programs ran in single core mode using 256MB of hash each. This way 6 games could be played at a time. The time control was set to 40 seconds for the whole game with a 2 second increment for each move.

The outcome was a 51.3% score for the version with the algorithm versus for that 48.7% without. More specifically, the algorithm secured 3284 wins, suffered 2972 losses and 5744 draws (so the draw rate was 47.9%). This corresponds to a performance difference of 9.0 Elo with a p-value, sometimes called the “likelihood of superiority”, of 99.996%.

6. Conclusions

It is demonstrated that upcoming repetition detection in software is not only feasible but that it can indeed be beneficial to a chess program with up to 9 Elo points shown.

The existing Zobrist framework, with an *XOR*-based combination of piece/square hashes, is sufficient as the basis for the algorithm. In fact, an addition-based hashing method could only increase the size of the tables by a factor of two.

The discovered relation between `S[0] ^ S[d]` and the exclusive *OR* of `S[0...d]`, for an even number of items, and modulo the side to move indicator, was new for the author.

The “no progress” pruning described by Hsu has not been implemented and tested.

Appendix. Supporting code for cuckoo table

Figure 8 gives the remaining definitions and initialization for the algorithm. As is normal in cuckoo hashing the initialization contains a potentially infinite loop. The probability of termination is high when the size of the table is at least twice the number of items added, which is the case here. The standard remedy against looping is to rehash and start over again. Here that could mean either changing the seed to the pseudo-random number used to create the underlying Zobrist hashes, or swapping or redefining the hash functions `H1` and `H2`.

```
zobrist_t Cuckoo[0x2000] // Cuckoo table with Zobrist hashes of valid reversible moves
int A[0x2000], B[0x2000] // The ‘move’ Cuckoo[i] is between squares A[i] and B[i]

#define H1(h) (((h)>>32) & 0x1fff) // First hash function for indexing the cuckoo tabl
#define H2(h) (((h)>>48) & 0x1fff) // Second hash function

// Test if the squares between a and b are all empty (a and b themselves excluded)
bool path_is_clear(int board[64], int a, int b) // No implementation here

// Test if move (a,b) is valid and reversible for the piece type on an empty board
bool is_valid_and_reversible_move(int piece, int a, int b) // No implementation here

#define Swap(x,y) ... // Macro to swap two variables. No implementation here

void init_cuckoo() // init_zobrist must have completed first
{
  // Create cuckoo table with reversible moves
  for (int piece=1; piece<=12; piece++)
    for (int a=0; a<64; a++)
      for (int b=a+1; b<64; b++) // a < b
        if (is_valid_and_reversible_move(piece, a, b)) {
          zobrist_t mv = ~(Zobrist[piece-1][a] ^ Zobrist[piece-1][b])
          int aa = a, bb = b
          int i = H1(mv)
          while (true) { // Insert in cuckoo table
            Swap(Cuckoo[i], mv)
            Swap(A[i], aa)
            Swap(B[i], bb)
            if (mv == 0) break // Arrived at empty slot so we’re done for this move
            i = (i == H1(mv)) ? H2(mv) : H1(mv) // Push victim to its alternative slot
          }
        }
}
```

Figure 8. Extended definitions and initialization

References

[1] *IBM’s Deep Blue Chess Grandmaster Chips*
F.-H. Hsu (1999)
IEEE Micro, March-April 1999,Vol. 19, pp. 70–81

[2] *Null Move and Deep Search*
Chr. Donninger (1993)
ICCA Journal, Vol. 16, No. 3, pp. 137-143

[3] *Cuckoo Hashing*
R. Pagh, F. F. Rodler (2001)
ESA 2001, Lecture Notes in Computer Science 2161, pp. 121-133