BITBOARD METHODS FOR GAMES

Cameron Browne[1]

QUT, Brisbane, Australia

ABSTRACT

Bitboards allow the efficient encoding of games for computer play and the application of fast bitwise-parallel algorithms for common game-related operations. This article describes: (1) a selection of bitboard techniques including an introduction to bitboards and bitwise operations, (2) a classification scheme that distinguishes *filter*, *query* and *update* methods, and (3) a sampling of bitboard algorithms for a range of games other than chess, with notes on their performance and practical application.

1. INTRODUCTION

Since the invention of the digital computer, programmers have been devising ingenious tricks to exploit their machine's architecture at the bitwise level to squeeze every drop of performance from the available resources. While processors have become faster over the decades and memory less restricted, there is still a need for such tricks in speed-critical applications, and a satisfaction in finding the most efficient algorithm for a given task.

Two primary resources on this topic are Warren's *Hacker's Delight* (2003) and Anderson's *Bit Twiddling Hacks* (1997), which each list a wealth of short code snippets for performing a range of operations efficiently using bitwise manipulations. Note the use of the term "hack" in both titles; such techniques are for programmers willing to get their hands dirty at the bitwise level, and are in many ways programming in its purest form.

1.1 Bitboards

Bitwise methods for programming games centre around the concept of the *bitboard*. This is a data structure designed for efficiently encoding game boards as sets of bits, first used for computer chess in the 1950s (Frey, 1977). Rather than allocating an integer for each board cell to store the value of any piece there, each cell is assigned a bit indicating the presence or absence of a piece (or pattern) there, requiring only a fraction of the memory. For example, the cells of an 8×8 chess board conveniently pack into a single 64-bit long integer.

Bitboards allow common game-related operations to be performed using fast bitwise manipulations. Pepicelli (2005) lists three main advantages of using bitboards.

1. *Memory Usage*: Bitboards encode the board state more efficiently than integer-per-cell encodings.
2. *Efficient Operation*: Read and write operations can be performed efficiently using bitwise operations.
3. *Bitwise-Parallel Operation*: Bitwise operations can be applied to all board cells simultaneously.

Efficient memory usage can be beneficial if it allows more operations to be performed from the (much faster) registers or cache. However, the potential for bitwise-parallel operation can also yield significant performance improvements. This means that game-specific calculations such as movement or win tests need only be applied once over the entire board in a bitwise-parallel manner, rather than individually for each cell or piece. Such operations are typically *stateless* as they operate equally over all cells with no prior knowledge about the board state, but can sometimes be optimised with the inclusion of state information.

[1]email:c.browne@qut.edu.au

1.2 Outline

Bitboard techniques for chess have been studied for over half a century and are now highly optimised and sophisticated.[2] There exist excellent resources on the topic, such as the *Chess Programming Wiki* (Lefler, 2014), and source code for some of the top programs is freely available.[3] However, many such optimisations remain the private property of their authors, especially for commercial programs, making it difficult to compile a comprehensive survey of bitboard techniques for chess.

In addition, this article presents a sample of bitboard techniques for games other than chess. The algorithms cover a range from basic techniques – which should be obvious to any programmer with the motivation to use bitboards in the first place – to more advanced techniques. The coverage is representative rather than exhaustive, as it would not be feasible to cover all bitboard methods in a single article, but will hopefully give some appreciation for the variety and usefulness of such approaches.

The following sections describe the encoding, definitions and operations used throughout the paper, then present a range of game-related bitboard algorithms organised according to the following classification scheme.

1. *Filters:* Bitsets derived from the current board state.
2. *Queries:* Numerical values derived from the current board state.
3. *Updates:* Manipulations of the current board state.

2. ENCODING

For our discussion, it is useful to draw on some concepts and terminology from digital morphology and image processing (Jain, 1989). Figure 1 shows four typical grids upon which board games are played.

1. *Linear*: Two directions of adjacency along one axis (2-connected).
2. *Square (Orthogonal)*: Four directions of adjacency along two axes (4-connected).
3. *Hexagonal*: Six directions of adjacency along three axes (6-connected).
4. *Square (Diagonal)*: Eight directions of adjacency along four axes (8-connected).

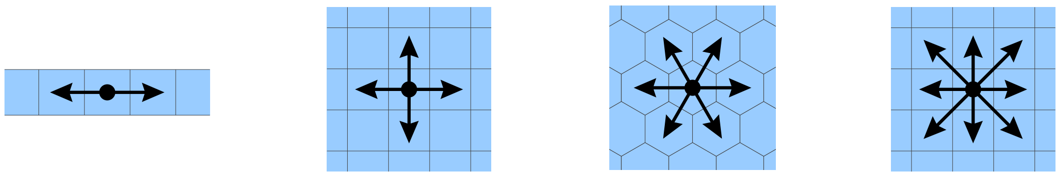

Figure 1: Typical game grids: linear (1 axis), square (2 axes), hexagonal (3 axes) and square (4 axes).

In the context of games, two cells are *adjacent* if a move from one cell to the other denotes a single step. More formally, two cells are adjacent if they form the end points of an edge in the board's underlying connectivity graph. The arrows in Figure 1 indicate directions of adjacency from a given cell.

2.1 Separating Bits

Each bit in a bitboard corresponds to a board cell, with rows typically laid end-to-end in memory. It can be convenient to separate each row with an off-board *separating bit*, to simplify the handling of boundary conditions and avoid adjacent rows being shifted into each other incorrectly. For example, Figure 2 shows a 5×5 square (orthogonal) board with a separating bit appended to each row (Fig. 2, top) and its corresponding bitboard mask (Fig. 2, bottom), with bits corresponding to on-board cells shaded.

[2]And also bitboard techniques for computer Go in more recent years.
[3]See, for example, the Stockfish chess engine by Costalba (2008).

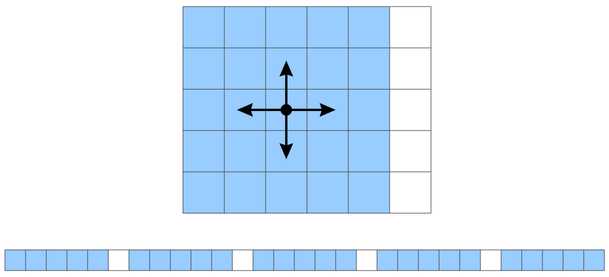

Figure 2: A 5×5 square (orthogonal) board with separating bits in the right hand column and its mask.

There is a tradeoff between the convenience of including separating bits and their cost; for example, an 8×8 chess board would no longer pack into a single 64-bit long integer. The alternative to using such separating bits is to mask out potential seepage with every operation, which can significantly increase the number of operations required to perform an algorithm, although increasing the number of words required to store the bitboard can also be detrimental. For simplicity, we assume the presence of separating bits in the following examples.[4]

We introduce the term $shift_d$ to describe the number of bit positions that must be right shifted in order to align adjacent bits in direction d. For the square (orthogonal) grid shown in Figure 2, the shift values in each direction $\{N, E, S, W\}$ would be $shift_N$ = *cols*+1, $shift_E$ = 1, $shift_S = -(cols + 1)$ and $shift_W = -1$, where *cols* is the number of board columns and a negative value implies a left shift.[5] Note that 1 is added to the N and S values to account for the separating bit.

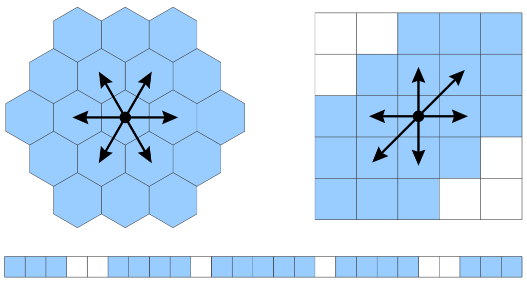

Figure 3: Staggered hexagonal encoding inherently separates each row in the bitboard.

Hexagonal boards can be staggered to pack into a square array, as shown in Figure 3. This is convenient for bitboard representation, as staggering inherently introduces separating off-board bits between rows. The three axes of the hexagonal grid require six shift directions: $shift_{NE} = cols + 1$, $shift_E = 1$, $shift_{SE} = -cols$, $shift_{SW} = -(cols + 1)$, $shift_W = -1$ and $shift_{NW} = cols$.

Bitboards can be *interleaved* such that a single bitboard contains the bits for all players, encoded as a sequence of n-bit tuples (see Subsection **??**). However, the following examples assume two players with a separate bitboard each, and a square (orthogonal) grid with separating bits, unless otherwise stated.

3. DEFINITIONS

The following terms are used throughout the paper.

- $bits_w$ = pieces belonging to White.
- $bits_b$ = pieces belonging to Black.
- $bits_p$ = pieces belonging to the current player.
- $bits_o$ = pieces belonging to their opponent.
- $bits_m$ = the cell at which the last move was played.

[4] A single separating bit handles single adjacent steps; if steps of distance 2 are expected then two separating bits are required, and so on.
[5] If using Java, be sure to use the unsigned right shift operator $>>>$ rather than $>>$ for right shifts.

- $mask$ = mask specifying the set of on-board cells (Figure 2).
- $mask_w$ = mask specifying the set of white cells in a checkerboard pattern (Figure 4, left).
- $mask_b$ = mask specifying the set of black cells in a checkerboard pattern (Figure 4, right).
- $|bits|$ = the *cardinality* of $bits$ (number of on-bits).
- $full$ = $bits_w \mid bits_b$ = the set of cells occupied by either player.
- $empty$ = $\sim full \;\&\; mask$ = the set of empty cells.
- $empty_p$ = $\sim bits_p \;\&\; mask$ = the set of cells not occupied by the current player.
- $empty_o$ = $\sim bits_o \;\&\; mask$ = the set of cells not occupied by their opponent.

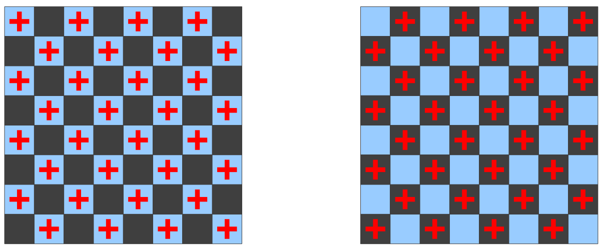

Figure 4: Checkerboard masks $mask_w$ and $mask_b$.

4. OPERATIONS

The fundamental bitwise operations can be categorised as either *logical* or *morphological* in nature, as follows.

4.1 Logical Operations

Figure 5 shows the standard logic operations that apply to bitboards.

Union ($|$)

The *union* of two bitboards is the combination of on-bits in both. For example, Figure 5 (top row) shows $bits_w$, $bits_b$ and their union $(bits_w \mid bits_b)$. On-bits are marked '+' at the corresponding cells.

Complement ($\sim$)

The *complement* of a bitboard flips each bit. Figure 5 (second row) shows $(bits_w \mid bits_b)$ and its complement, which is the set of empty cells. Note that no off-board bits are included, i.e. the mask shown in Figure 2 has been implicitly applied to the result.

Exclusive Or ($\wedge$)

The *exclusive or* (*xor*) of two bitboards sets bits that are on in one but not both. For example, Figure 5 (third row) shows the xor of $bits_w$ and the set of empty cells. The same result can be be achieved with $(a \mid b) \;\&\; \sim(a \;\&\; b)$, but xor is included for convenience and efficiency.

Intersection ($\&$)

The *intersection* of two bitboards gives the overlap of on-bits. For example, Figure 5 (bottom row) shows the intersection of $(bits_w \mid bits_b)$ with the xor result previously obtained.

4.2 Morphological Operations

The following two operations, shown in Figure 6, are drawn from digital morphology and image processing.

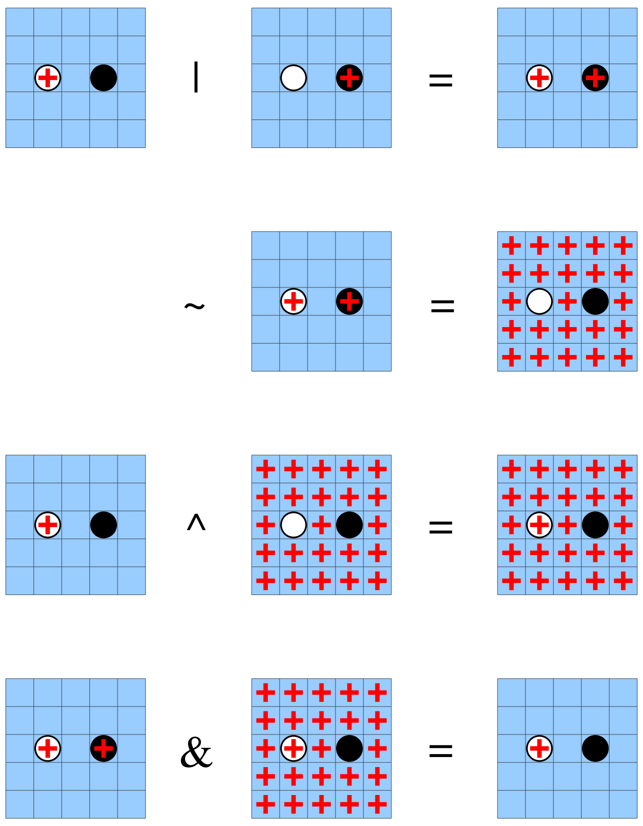

Figure 5: Logical bitwise operations: union ($|$), complement ($\sim$), xor ($\wedge$) and intersection ($\&$).

4.3 Dilation $\oplus$

Dilation is the expansion of on-bits by a given pattern, which for game-related tasks is typically an expansion to adjacent neighbours. For example, Figure 6 (top row) shows the dilation of $(bits_w \mid bits_b)$ to adjacent neighbours. A dilation in direction d is given by:

$$bits \ \oplus \ d \ = \ bits \mid (bits >> shift_d)$$

and a dilation one step in all square (orthogonal) directions is given by:

$$bits \ \oplus \ 1 \ = \ bits \mid (bits >> shift_N) \mid (bits >> shift_E) \mid (bits >> shift_S) \mid (bits >> shift_W).$$

Note that $\oplus$ is the symbol for dilation from digital morphology, and is not the bitwise operator $\oplus$ described by Knuth (2009), p2.

4.4 Erosion $\ominus$

Erosion shrinks the on-bits in each direction, and can conceptually be seen as the reverse operation of dilation. For example, Figure 6 (bottom row) shows the set of empty cells eroded by one step in each adjacent direction. Note that the two surviving on-bits were the only ones without adjacent off-bit neighbours (off-board cells are considered as off-bits for these calculations). An erosion in direction d is given by:

$$bits \ \ominus \ d \ = \ bits \ \& \ (bits >> shift_d)$$

and an erosion one step in all square (orthogonal) directions is given by:

$$bits \ \ominus \ 1 \ = \ bits \ \& \ (bits >> shift_N) \ \& \ (bits >> shift_E) \ \& \ (bits >> shift_S) \ \& \ (bits >> shift_W).$$

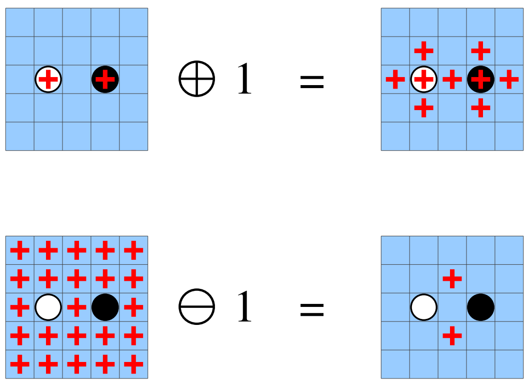

Figure 6: Morphological bitwise operations: dilation $\oplus$ and erosion $\ominus$.

5. ALGORITHMS

The following section describes a variety of practical game-based algorithms using bitwise operations, with pseudocode or actual code listed as appropriate. The algorithms are categorised as being either *filters* (5.1), *queries* (5.2) or *updates* (5.3). Filters and queries are *immutable* as they only read from the board bits, whereas updates are *mutable* as they also write to the board bits. Mutability has implications for parallelisation and thread safety.

All timings mentioned are based on tests implemented in Java 1.6 and run on a single thread of a standard laptop machine with 2 GHz $i7$ processor.

5.1 Board Filters

Board filters read from the current board state to produce new bitboards that satisfy certain criteria. Typical applications include determining subsets of board cells, for example, when generating legal moves. Board filters are typically immutable. We distinguish between *neighbour filters* (5.1.1), *line move filters* (5.1.2) and *pawnlike move filters* (5.1.3).

5.1.1 Neighbour Filters

Algorithm 1 detects the empty neighbours of both players by intersecting the dilation of all pieces with the set of empty cells. This process is demonstrated for the board position shown in Figure 7.

Algorithm 1 Empty Neighbours of Players

```
1. $nbors = (full \oplus 1)\ \&\ empty$
```

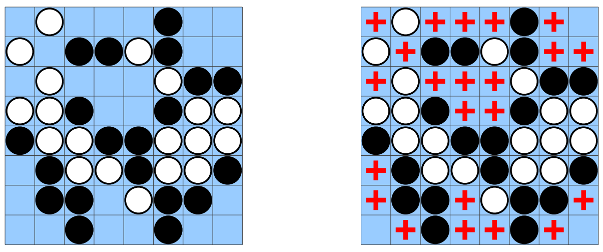

Figure 7: Empty neighbours of both players.

Algorithm 2 performs the more specific calculation of empty neighbours for a particular player in each direction. For example, Figure 8 shows White's empty neighbours in each direction.

Algorithm 2 Empty Neighbours of a Player in Each Direction

```
1. for each direction $d$
2.     $nbors_d = (bits_p >> shift_d) \& empty_p$
```

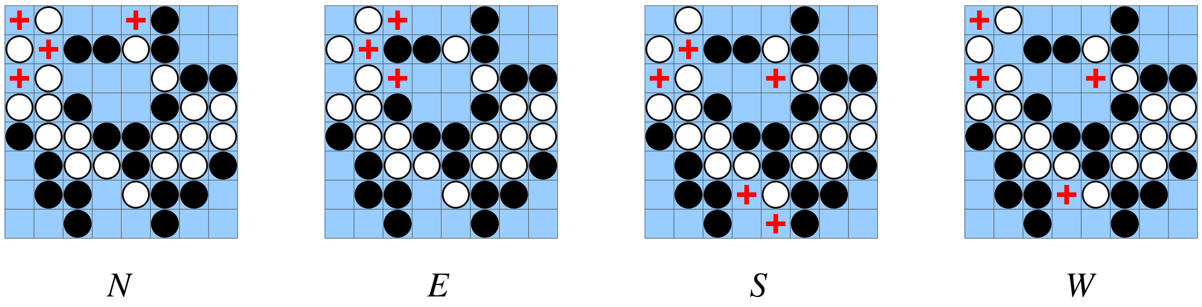

Figure 8: Empty neighbours of White in each direction.

Algorithm 3 detects enemy neighbours for a particular player in each direction, as shown in Figure 9 for White.

Algorithm 3 Enemy Neighbours of a Player in Each Direction

```
1. for each direction $d$
2.     $nbors_d = (bits_p >> shift_d) \& bits_o$
```

This approach was used for the game of Clobber, in which players must move one of their pieces to capture an adjacent enemy piece each turn, to generate all legal move destinations in four bitwise-parallel calculations (one for each direction). The "from" cell for each move can be easily reconstructed as the white piece that must exist adjacent to the destination cell in the opposite direction. This approach gave a significant speedup over a non-bitwise implementation to allow around 200,000 random playouts per second on a full 8×8 board.

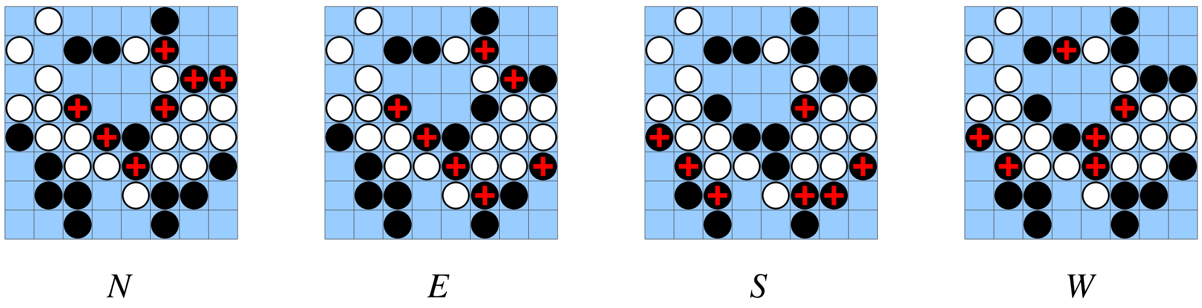

Figure 9: Enemy neighbours of White in each direction.

5.1.2 Line Move Filters

The game Othello (aka Reversi) is played on an 8×8 square (diagonal) board, and moves are only allowed at empty cells that would capture a line of enemy pieces (in any direction) by capping the line with a friendly piece at each end. Algorithm 4 generates such *line cap* moves in a bitwise-parallel manner.[6]

[6]Based on code from: https://github.com/EivindEE/Reversi

Algorithm 4 Line Cap Moves

```
1. $moves = \emptyset$
2. **for each** direction $d$
3.   $candidates = bits_o \,\&\, (bits_p >> shift_d)$
4.   **while** $candidates \,! = \emptyset$
5.     $moves \;|= empty \,\&\, (candidates >> shift_d)$
6.     $candidates = bits_o \,\&\, (candidates >> shift_d)$
```

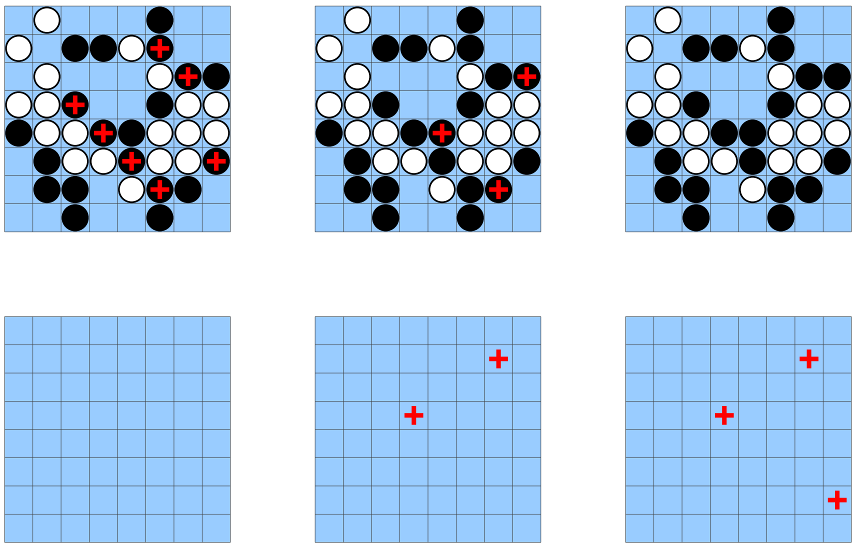

Figure 10: Line cap moves in the E direction: line propagation (top) and legal moves (bottom).

For each adjacent direction, a set of candidate moves is generated by the intersection of enemy pieces with friendly pieces shifted one step in that direction (line 3). While this candidate set is not empty, it is shifted one step in the current direction and intersected with the empty set to give any legal moves (line 5) and overwritten with its intersection with the enemy piece set to progress a step (line 6).

Figure 10 shows this process applied to white pieces in the E direction. The top row shows the initial candidate set (left) propagated one step (middle) then empty on the next step (right). The bottom row shows the legal move set growing with each step; a white piece at any of these cells would capture a horizontal black line.

5.1.3 Pawnlike Move Filters

Breakthrough[7] is a modern board game played on a square (diagonal) board, in which pieces move either: (1) straight or diagonally forward to an adjacent empty cell, or (2) diagonally forward to capture an adjacent enemy piece. Algorithm 5 generates legal move destinations for Breakthrough by intersecting the player's piece set, shifted in the appropriate forward directions, with the empty cell set (line 3) and then the enemy piece set (line 5). Figure 11 shows the resulting move destinations for White in the given position.

Algorithm 5 Breakthrough Moves

```
1. $moves = \emptyset$
2. **for each** forward direction $d$
3.   $moves \;|= empty \,\&\, (bits_p >> shift_d)$
4. **for each** forward diagonal direction $d$
5.   $moves \;|= bits_o \,\&\, (bits_p >> shift_d)$
```

[7]http://en.wikipedia.org/wiki/Breakthrough_(board_game)

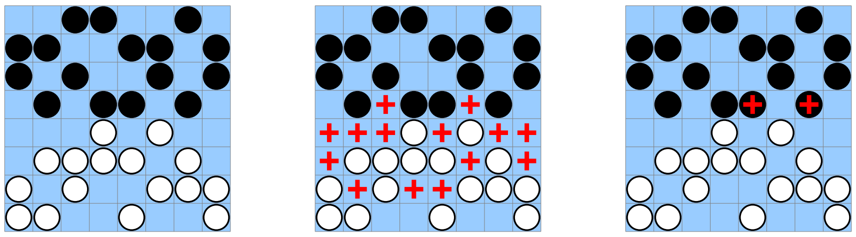

Figure 11: A Breakthrough position, moves to empty cells and capture moves.

Note that the generated move destinations are ambiguous, as different pieces could move to the same destination. However, this information is still useful for quickly detecting attacks, for move planning purposes.

5.2 Board Queries

Board queries read from the current board state to return a value, typically numerical or Boolean, e.g. for win or movement tests. Board queries are typically immutable. We distinguish between *line queries* (5.2.1), *piece count queries* (5.2.2), *connection queries* (5.2.3) and *pattern queries* (5.2.4).

5.2.1 Line Queries

Lines of pieces can be detected by iteratively eroding the player's bits along each axis and counting the maximum number of erosions, as summarised in Algorithm 6. The term $shift_a$ refers to the forward (positive) direction along each axis; only one direction needs to be tested per axis.

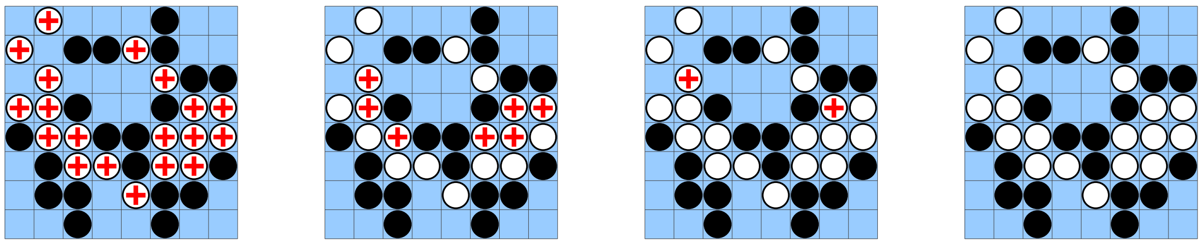

Figure 12: Erosion in the N direction detects two vertical white lines of length 3.

Algorithm 6 Line Detection

```
1. return argmax $length_a$
2.   for each axis $a$
3.     $bits = bits_p$
4.     $length_a = 0$
5.     while $(bits = bits \,\&\, (bits >> shift_a)) \,!= \emptyset$
6.       $length_a$++
```

Figure 12 shows the detection of two vertical white lines of length three by iteratively eroding $bits_w$ in the N direction, and Figure 13 shows the detection of a single horizontal white line of length three by iteratively eroding $bits_w$ in the E direction; White's maximum line count is length three.

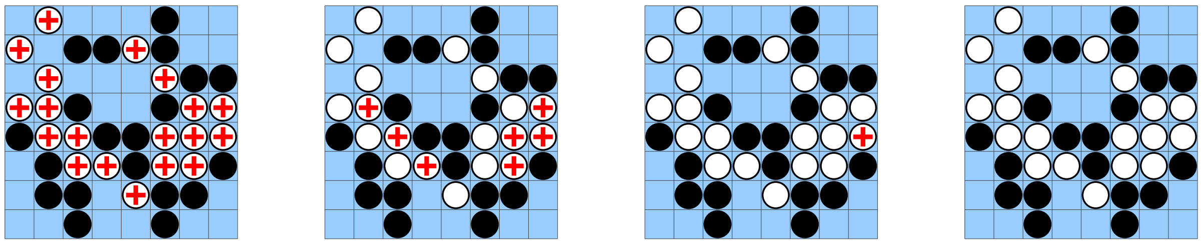

Figure 13: Erosion in the E direction detects a horizontal white line of length 3.

This approach was used for win/loss detection in Yavalath, an N-in-a-row game played on a hexagonal grid in which 4-in-a-row wins but 3-in-a-row loses, to give an approximately 2$\times$ speedup over a non-parallel neighbour-following approach and allow over 500,000 random playouts per second on a full 61-cell board (Browne and Maire, 2014).

For the game of Connect Four, an N-in-a-row game in which the target line length is always four (or more), line detection can be further optimised as shown in Algorithm 7[8]:

Algorithm 7 Line of 4 Detection

```
1. for each axis $a$
2.     $pairs = bits_p \& (bits_p >> shift_a)$
3.     if $pairs \& (pairs >> shift_{aa}) \mathrel{!}= \emptyset$
4.         return true
5. return false
```

where $shift_{aa}$ is a double step in direction a. This algorithm finds adjacent pairs of pieces belonging to the specified player along axis a in line 2, then adjacent pairs of pairs (i.e. 4-in-a-row) in line 3. Again, only one direction needs to be tested along each axis.

5.2.2 Piece Count Queries

The number of pieces belonging to player p is given by $|bits_p|$. The number of pieces on cells of colour c is given by $|\ bits_p \ \& \ mask_c\ |$, as shown in Figure 14.

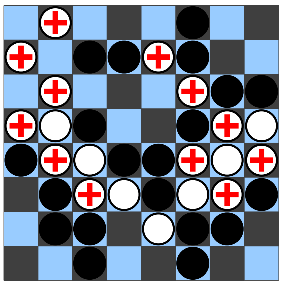

Figure 14: There are twelve white pieces on black cells.

Efficient methods for counting the on-bits in an integer can be found in Warren Jr (2003) and Anderson (1997).

5.2.3 Connection Queries

In connection games such as Hex and Y, players strive to connect edges of the board with connected sets of pieces of their colour. Connection tests in such games are typically performed using a *union find* approach (Sedgewick

[8]See: http://en.wikipedia.org/wiki/Bitboard

and Wayne, 2011), which tracks group membership and iteratively updates this information as groups meet.

However, bitwise-parallel methods offer a fast stateless alternative using the *Y reduction* rule (van Rijswijck, 2002), which iteratively reduces triplets of cell values to their majority value, as shown in Figure 15. This majority value represents a guaranteed connection through the triplet, hence any board that reduces to a single cell of a player's colour indicates a win for that player. The reduction value is 0 (empty) if there is no clear majority colour, which can occur if any members of the triplet are empty.

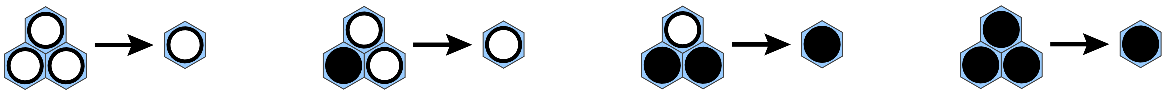

Figure 15: The Y reduction rule reduces triplets of cell values to their majority value.

For example, Figure 16 shows a completed game of Y won by Black, who has connected all three board sides with a chain of black pieces. This can be verified by reducing the position to a single black cell.

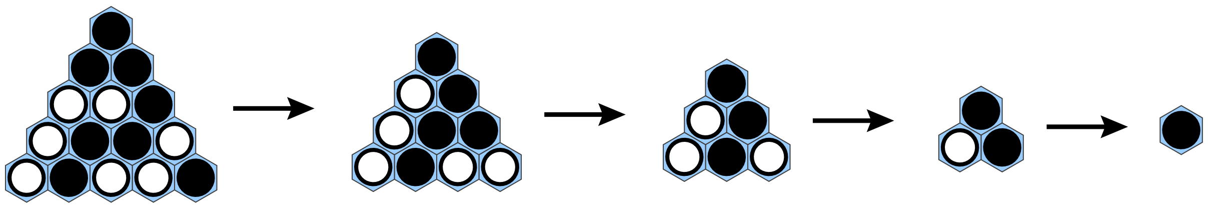

Figure 16: Y reduction proves a black connection between all three sides.

Listing 1 shows a Java implementation of bitwise-parallel Y reduction, from Browne and Tavener (2012a). In this case the bitboard is split into a separate integer for each column, and each column interleaved such that board cells are described by two consecutive bits (`00` = empty, `01` = white, `10` = black, `11` = unused), which simplifies the reduction calculation. The actual code listing is given in this case as it is just as concise as a pseudocode description would be, highlighting the elegance of bitwise approaches.

```
int reduce(int[] bits)
{
    for (int pass = 0; pass < cols-1; pass++)
        for (int col = 0; col < cols-1-pass; col++)
        {
            final int a = bits[col];
            final int b = (a >> 2);             // NE neighbours
            final int c = bits[col+1];          // E neighbours
            bits[col] = (a & (b | c)) | (b & c);   // reduce {a, b, c} triplets
        }
    return bits[0] & 0x3;    // value at apex is winner's colour, else 0 if no winner
}
```

Listing 1: Java implementation of bitwise-parallel Y reduction.

For each reduction pass, the algorithm reduces triplets of cells simultaneously in a bitwise-parallel manner within a decreasing number of columns. This provided an approximately 50× speedup over union find win detection for Y, for cases in which board positions must be tested without state information, e.g. following random board fills in Monte Carlo simulation (Browne and Tavener, 2012a). However, there is a tradeoff between the speed advantage of bitwise-parallel reduction, which allows very fast "light" playouts, and the slower but more realistic results given by incorporating domain knowledge into the playouts on a move-by-move basis using "heavy" playouts.

5.2.4 Pattern Queries

An obvious use of bitboards is to encode and detect target patterns. Such patterns can be pre-calculated to a database of n constants, notated $bits_{1...n}$, then occurrences can be found quickly at run-time by intersecting patterns with the board state as per Algorithm 8.

Algorithm 8 Pattern Matching

```
1. for each pattern $1 \ldots n$
2.     if $(bits_n \ \& \ bits_p) \ == \ bits_n$
3.         return $n$
4. return $0$
```

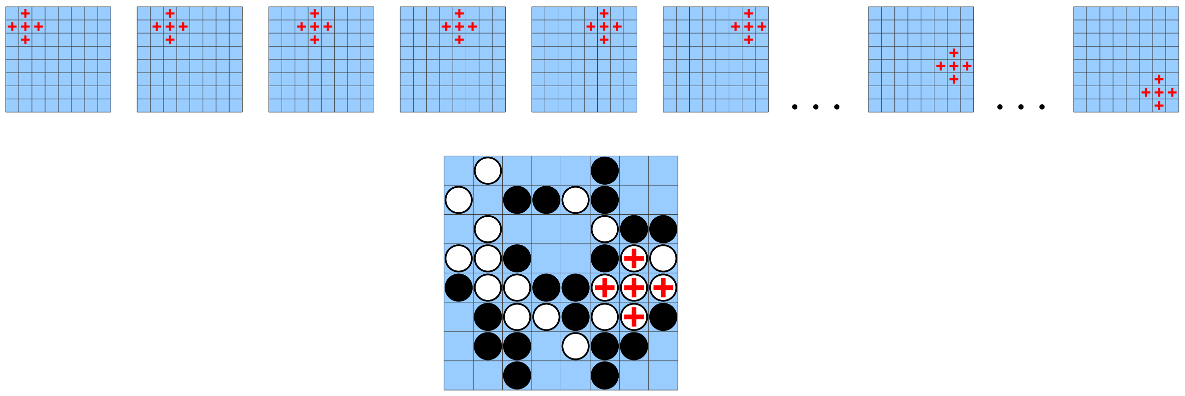

Figure 17: The white pieces indicated match a target pattern.

For example, Figure 17 shows a game in which players must achieve a target '+' pattern with their pieces. The top row shows some of the 36 possible patterns, and the bottom row shows a match with $bits_w$.

The target behaviour is easily customised by modifying the test in line 2, for example, the following test would only detect piece clusters that match patterns *exactly*, and would not match the pattern shown in Figure 17.

$$\textbf{if} \ (bits_n \ \& \ bits_p) \ == \ bits_n \ \ \&\& \ \ ((bits_n \oplus 1) \ \& \ \sim bits_n \ \& \ bits_p) \ == \ \emptyset$$

Other variations might include matching two or more target patterns (with or without overlap), matching two or more different types of target patterns (with or without overlap), patterns that adjoin without overlap, cells at which different patterns overlap, etc. State information can make the process more efficient, for example, storing references to patterns according to which cells they intersect, then for each move only testing those patterns intersecting the cells that are affected.

5.3 Board Updates

Board updates modify the current board state to achieve a desired result. Typical applications include applying moves or biasing Monte Carlo playouts. Board updates are mutable in nature. We distinguish between *flip updates* (5.3.1), *capture updates* (5.3.2), *cellular automata updates* (5.3.3) and *packing updates* (5.3.4).

5.3.1 Flip Updates

Algorithm 9 shows a bitwise-parallel algorithm for flipping all adjacent enemy pieces to a player's colour. The player's pieces $bits_p$ are dilated and intersected with the enemy opponent's pieces $bits_o$, then the colour conversion is performed by removing the resulting bits from $bits_o$ and adding them to $bits_p$. This process is shown in Figure 18 from White's perspective.

Algorithm 9 Flip All Enemy Neighbours

```
1. $flip \ = (bits_p \oplus 1) \ \& \ bits_o$
2. $bits_o \ \&= \sim flip$
3. $bits_p \ |= \ flip$
```

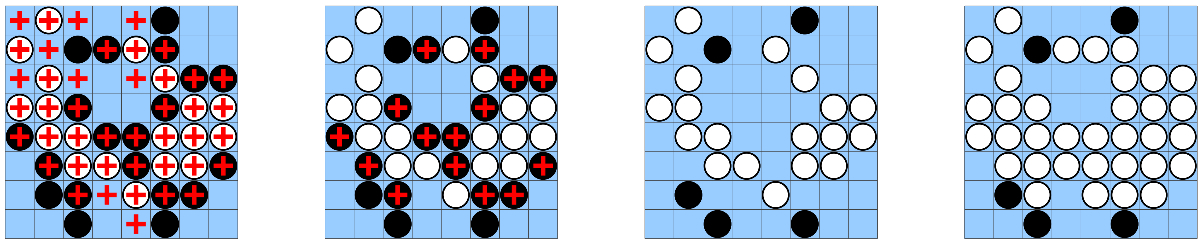

Figure 18: Enemy neighbours of White flipped (i.e. removed and replaced) to white.

This example is rather harsh, as most black pieces were flipped; a more realistic requirement would be that only enemy pieces adjacent to the player's last move are flipped. This process is described in Algorithm 10 and shown in Figure 19 relative to the last White move, the location of which is given by $bits_m$. This is the same as Algorithm 9 except that the initial dilation is limited to neighbours of the last move.

Algorithm 10 Flip Enemy Neighbours of Last Move

```
1. $flip \quad = (bits_m \oplus 1)\ \&\ bits_o$
2. $bits_o \ \&= \sim flip$
3. $bits_p \ |= \ flip$
```

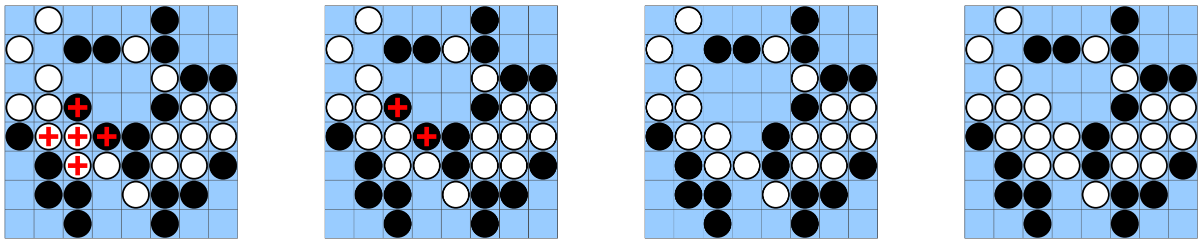

Figure 19: Enemy neighbours of the last White move flipped (i.e. removed and replaced) to white.

Algorithm 11 shows the similar process of toggling the values of a selected cell and its immediate neighbours on or off each turn. This mechanism is used in puzzle games such as Lights Out,[9] in which the aim is to turn all cells off from a given state. For example, Figure 20 shows a sequence of moves that toggles all the black pieces $bits_b$ off, where each move consists of xoring the bit set given by a cell and its immediate neighbours $(bits_m \oplus 1)$.

Algorithm 11 Invert Neighbourhood of Move

```
1. $bits_b \ \wedge = \ bits_m \oplus 1$
```

5.3.2 Capture Updates

Algorithm 12 performs surround capture, i.e. the removal of connected sets of pieces of a specified colour with no *freedom* (adjacent empty cells).

Figure 21 shows this process relative to White. Firstly, the set of empty cells is dilated and intersected with $bits_w$ to give the initial set of white pieces with freedom (line 1, left). This free set is then iteratively dilated and intersected with $bits_w$ (lines 3 – 5) to propagate freedom to two more pieces... and propagated again to two more pieces... until no further propagation occurs. The player's bits are then replaced by the result (line 6), eliminating any pieces without freedom (right).

[9]http://mathworld.wolfram.com/LightsOutPuzzle.html

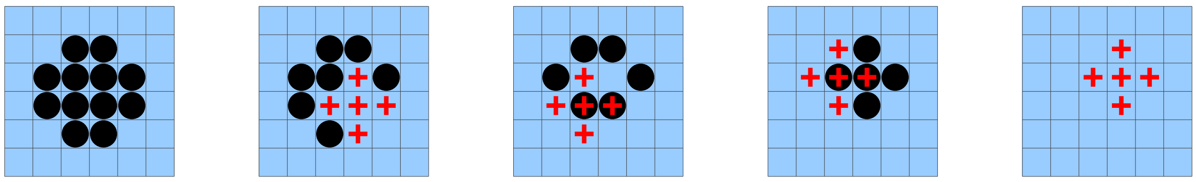

Figure 20: Sequence of neighbourhood inversions to turn all cells off (Lights Out).

Algorithm 12 Surround Capture

```
1. $free \ = \ (empty \ \oplus \ 1) \ \& \ bits_p$
2. $result \ = \ \emptyset$
3. while $result \ != \ free$
4.     $result = free$
5.     $free \ = (free \ \oplus \ 1) \ \& \ bits_p$
6. $bits_p = result$
```

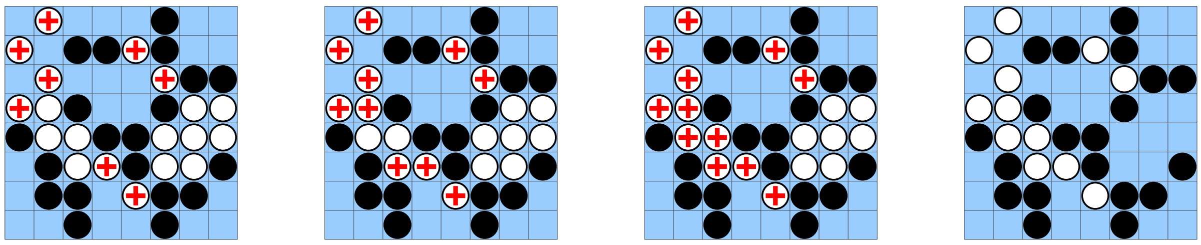

Figure 21: White freedoms propagate to neighbours, then freedomless groups are eliminated.

An obvious application of this approach is to the game of Go. Similar approaches are used in current Go programs, for example, BitmapGo[10] which uses a recursive bitwise-parallel approach radiating from the (up to) four connected neighbours of the last move played. However, the exact algorithm outlined in Algorithm 12 is not used and is unlikely to benefit current Go programs, due to the additional calculation required to detect self-suicide moves and *superko* cases (the repetition of previous board states), and the fact that a significant amount of state information must typically be incorporated into Go playouts to achieve strong play.

Similar approaches could be used for querying all liberties for a given board state (Line 1 of Algorithm 12), or liberties of a connected group of pieces, by dilating that group and intersecting the result with the empty cell set.

5.3.3 Cellular Automata Updates

Long Life is an 8×8 version of Conway's Game of Life (Gardner, 1970) with edge wraparound. The Game of Life is a cellular automata "solitaire game" in which cells within a grid live or die in the subsequent generation depending on the count of live neighbours. In the standard game, dead cells with three live neighbours and live cells with two or three live neighbours live in the next generation, and all other cells die.

For example, Figure 22 shows one cycle of a pattern called the *glider* in Long Life. The glider pattern starts in the middle of the grid (top left), then shifts down and to the right over subsequent generations, wrapping around the board edges to return to its starting position after 32 iterations on the 8×8 board.

This version of the game is called Long Life as it is implemented in a single 64-bit long integer for bitwise-parallel operation (Browne and Tavener, 2012b). Each bit corresponds to a cell in the 8×8 board, and no separating bits are required as wraparound is allowed. Appendix A shows Java code for iterating the entire board one generation in fewer than 100 logical operations, making it more than 100× faster than a naïve (non-bitwise) implementation.

[10]The BitmapGo source code is available at: http://comments.gmane.org/gmane.games.devel.go/20854

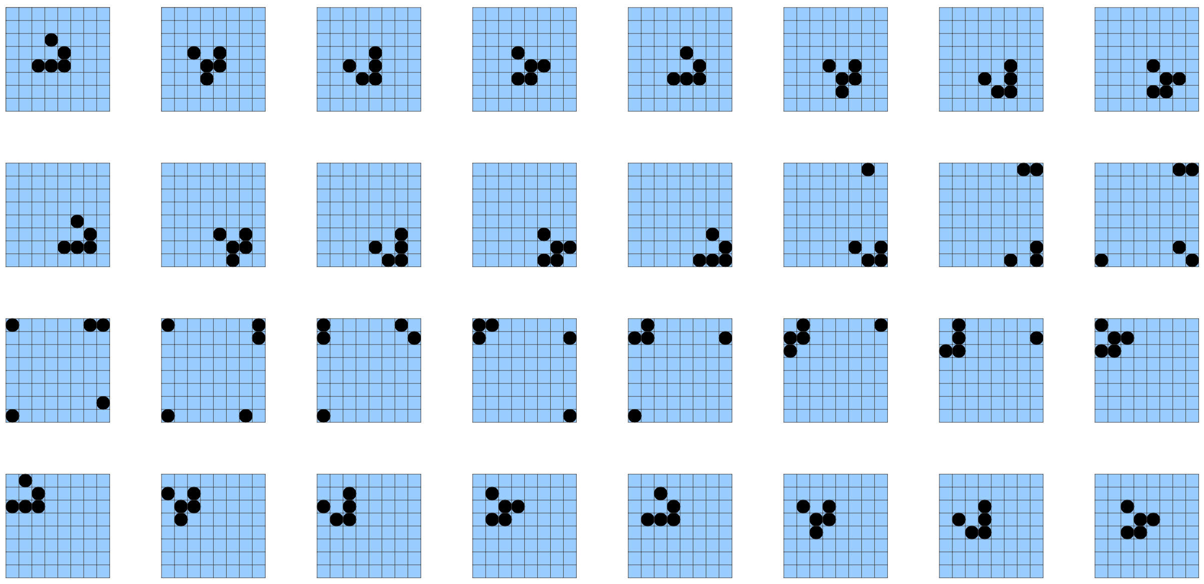
Figure 22: One cycle of a glider in Long Life.

A variable $c11$ describes the current board state and three variables $bit1$, $bit2$ and $bit3$ act as bit plane adders that accumulate neighbour counts in a bitwise-parallel manner over all cells simultaneously, through bit shifts and overflow carries as needed. The following expression then simultaneously calculates all live cells in the next generation, based on the accumulated neighbour tallies (see Appendix A for details).

$$((c11 \mid bit1) \,\&\, bit2 \,\&\, {\sim}bit3)$$

An alternative bitwise-parallel approach for boards larger than 8×8 is described in Pepicelli (2005).

5.3.4 Packing Updates

Puzzles that involve packing pieces into an area (e.g. Pentominoes) or volume (e.g. the Soma Cube or Bedlam Cube) can be solved efficiently using bitboards, as described in Algorithm 13.

Algorithm 13 Puzzle Packing

```
1.  function $solve()$
2.      $pack(\emptyset, \emptyset)$
3.  function $pack(bits, used)$
4.      if $bits == mask$
5.          return   $//\, packing\ found$
6.      $cell \leftarrow$ next empty cell
7.      for each piece $u$ not in $used$
8.          $used_u \leftarrow used \mid u$
9.          $critical \leftarrow {\sim}covered(bits, {\sim}used_u) \,\&\, mask$
10.         for each placement $pl_u$ of piece $u$ containing $cell$
11.             if $(pl_u \,\&\, bits) == \emptyset \,\&\&\, (pl_u \,\&\, critical) == critical$
12.                 $bits_u \leftarrow bits \mid pl_u$
13.                 if $(covered(bits_u, {\sim}used_u) \mid bits_u) == mask$
14.                     if $(playable(bits_u, {\sim}used_u) == used_u$
15.                         $pack(state_u, used_u)$
```

For this algorithm, all possible placements of each piece at all rotations and translations within the board are pre-generated, and stored as a collection of bitboards indicating the cells that each placement occupies. The algorithm also assumes the existence of two functions: $covered(bits, pieces)$ which returns a bitboard describing the cells covered by all possible placements of all specified pieces in the given board state, and $playable(bits, pieces)$

which returns the set of pieces that can be played in the given board state. In each case $pieces$ is a bitset indicating the presence/absence of each piece.

The algorithm recursively tries to pack pieces in the next available empty cell (line 6), which is given by the least significant off-bit. For each unused piece u (line 7) a set of $critical$ cells is determined (line 9), which is the set of cells not covered by any placement of any other remaining piece; u must occupy these cells. For each placement pl_u of u, if pl_u does not intersect any pieces already placed and covers the critical set (line 11) then it is added to the board (line 12). If the remaining pieces cover the remaining empty cells (line 13) and all remaining pieces have at least one legal placement (line 14), then all is well and the algorithm recurses to the next empty cell.

A Java implementation of this algorithm finds all 11,520 packings of the Soma Cube (and several Bedlam Cube solutions) within half a second, which is comparable to existing optimised C++ discrete packing solvers.[11]

6. CONCLUSION

Bitboard methods allow the efficient implementation of simple game-related algorithms that minimise memory usage and can yield significant speed increases. They are especially beneficial in cases where state information leading up to a board position is not needed, and the desired calculation can be applied to all cells simultaneously in a bitwise-parallel manner. The range of examples presented in this article hopefully demonstrate the versatility and usefulness of bitboard methods for a range of games.

However, they are not a panacea for all game-related tasks. Advanced chess programs will often maintain dual bitboard and non-bitboard representations of the board state and use whichever is most appropriate for a given task, allowing the best of both worlds. But the efficient memory usage and inherent parallelism of bitwise approaches makes them beneficial in many situations, and potentially attractive candidates for GPU implementation.

ACKNOWLEDGMENTS

Thanks to Stephen Tavener and Greg Schmidt for their suggestions, and members of the Computer-Go mailing list (computer-go@dvandva.org) for their advice on Go-related aspects. Also thanks to Jaap van den Herik for his excellent editorial comments and the anonymous reviewers for their suggestions that helped improve the article.

This work was funded by a Vice-Chancellor's Research Fellowship from the Queensland University of Technology (QUT), Brisbane, Australia, as part of the project *Games Without Frontiers*.

7. REFERENCES

Anderson, S. E. (1997). Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html.

Browne, C. and Maire, F. (2014). A Monte Carlo Resistant Game. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6 (submitted).

Browne, C. and Tavener, S. (2012a). Bitwise-Parallel Reduction for Connection Tests. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, pp. 112–119).

Browne, C. and Tavener, S. (2012b). Life in the Fast Lane. *AIFactory Newsletter*. http://www.aifactory.co.uk/newsletter/2012_02_fast_lane.htm.

Costalba, M. (2008). Stockfish. https://github.com/official-stockfish/Stockfish.

Frey, P. W. (ed.) (1977). *Chess Skill in Man and Machine*. Springer-Verlag, New York, NY.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, Vol. 223, pp. 120–123).

Jain, A. (1989). *Fundamentals of Digital Image Processing*. Prentice-Hall, Englewood Cliffs, NJ.

[11] See, for example, Bruce Cropley's BCSolve: http://www.fam-bundgaard.dk/SOMA/NEWS/N031026.HTM

Knuth, D. (2009). *The Art of Computer Programming, Vol. 4, Fascicle 1: Bitwise Tricks & Techniques.* Addison Wesley, Upper Saddle River, NJ.

Lefler, M. (2014). Bitboards. *Chess Programming Wiki.* https://chessprogramming.wikispaces.com/Bitboards.

Pepicelli, G. (2005). Bitwise Optimization in Java: Bitfields, Bitboards, and Beyond. http://www.onjava.com/pub/a/onjava/2005/02/02/bitsets.html?page=2.

Rijswijck, J. van (2002). Search and evaluation in Hex. Technical report, University of Alberta, Edmonton.

Sedgewick, R. and Wayne, K. (2011). *Algorithms.* Addison Wesley, Boston, MA, fourth edition.

Warren Jr, H. S. (2003). *Hacker's Delight.* Addison-Wesley, Upper Saddle River, NJ.

APPENDIX A. FAST LIFE ALGORITHM

The following listing shows Java code for a fast bitwise-parallel implementation of Conway's Game of Life on an 8×8 grid with edge wraparound (Browne and Tavener, 2012b).

```java
// Edge cell masks (Left/Right/Top/Bottom)
static final long L = 0x8080808080808080L;
static final long R = 0x0101010101010101L;
static final long T = 0x00000000000000FFL;
static final long B = 0xFF00000000000000L;
static final long NL = ~L;
static final long NR = ~R;
long bit1, bit2, bit3;  // bit plane adders

// Add neighbour count cXX to bit registers
void add(long cXX)
{
    long carry1 = bit1 & cXX;
    long carry2 = bit2 & carry1;
    bit1 ^= cXX;
    bit2 ^= carry1;
    bit3 |= carry2;
}

// Perform one step of 8x8 Life B3/S23 on state c11
long step(long c11)
{
    // Shift nbors into position, with wrap
    long c10 = c11 >>> 8 | ((c11 & T) <<  56);
    long c12 = c11 <<  8 | ((c11 & B) >>> 56);
    long c00 = (c10 & NL)<< 1 | ((c10 & L)>>>7);
    long c01 = (c11 & NL)<< 1 | ((c11 & L)>>>7);
    long c02 = (c12 & NL)<< 1 | ((c12 & L)>>>7);
    long c20 = (c10 & NR)>>>1 | ((c10 & R)<< 7);
    long c21 = (c11 & NR)>>>1 | ((c11 & R)<< 7);
    long c22 = (c12 & NR)>>>1 | ((c12 & R)<< 7);

    // Reset the bit registers
    bit1 = 0;  bit2 = 0;  bit3 = 0;

    // Accumulate live neighbour counts
    add(c00);  add(c01);  add(c02);  add(c10);
    add(c12);  add(c20);  add(c21);  add(c22);

    // Return live cases
    return ((c11 | bit1) & bit2 & ~bit3);
}
```

Listing 2: Java implementation of Long Life.

APPENDIX B. SCALABILITY

Bitboard approaches give best performance when the entire bitboard packs into a single integer, as with chess. However, this is rarely possible in non-trivial real-world cases. One way to handle larger boards is to split the bitboard into columns as per the Y example (Subsection 5.2.3), but a more efficient solution is to pack the bits into as few consecutive integers as possible. For example, Figure 23 shows the bitboard for one player for a 19×19 square board (including separating bits) spread over six 64-bit long integers.

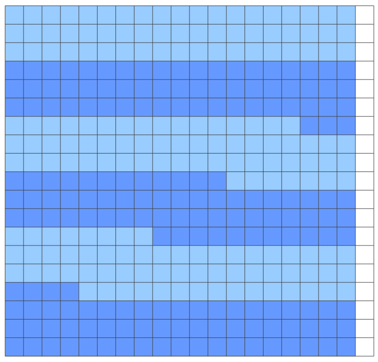

Figure 23: A 19×19 square board requires six 64-bit long integers per player.

Bitboards spread over multiple integers require careful handling of boundary cases, so that bits align with the appropriate bits in the previous or next integer when shifted across their common boundary. Such boundary cases can be handled by repeating the bitwise calculation applied to each integer to the previous and next integers, shifted by the appropriate amount. For example, if the bitboard integers are numbered $bits_{1...I}$ then for $bits_i$:

```
if $(i > 0)$ include $(bits_{i-1} >> shift_w)$ in the calculation
if $(i < I - 1)$ include $(bits_{i+1} >> shift_w)$ in the calculation
```

where $shift_w = B - shift_d$ is the shift size required to wrap the appropriate bits from the "far" side of the previous or next integer (B is the base data size or number of bits in each integer). If no shift amount exceeds B, then only the previous and next integers need be handled.