An Efficient Chinese Checkers Implementation: Ranking, Bitboards, and BMI2 pext and pdep Instructions

Nathan R. Sturtevant[1,2][0000−0003−4318−2791]

[1] Department Computing Science, University of Alberta, Canada
[2] Canada CIFAR Chair, Alberta Machine Intelligence Institute
nathanst@ualberta.ca

Abstract. The game of Chinese Checkers has a computationally expensive move generation function. Finding legal moves dominates the performance of a Chinese Checkers program. This paper describes a bitboard representation of the Chinese Checkers board, how to efficiently generate and apply moves to the board, and how to rank and unrank states. When available, the BMI2 PDEP (parallel bits deposit) and PEXT (parallel bit extract) instructions offer significant efficiency gains, especially over a non-bitboard based implementation.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

One important feature of top-performance game playing programs is the ability to search the game tree efficiently. In many games this has meant using efficient bitwise operations on bitboards for the game state [1–3, 5–7].

Thus, while bitboard representations are common, there is, to our knowledge, no bitboard description for Chinese Checkers found in the literature. Chinese Checkers has received research for many years [9, 10], but has received less attention than other games.

This paper describes how bitboards can be implemented for Chinese Checkers. In particular, beyond the more routine bitwise operations that have commonly been used in other games, this paper describes how the BMI2 `pext` and `pdep` operations can be used to extract and deposit bits to allow for efficient move generation in Chinese Checkers.

In addition to describing our efficient implementation, we provide experimental results showing the performance of the approach. For most operations it is 2-3x faster than our baseline implementation. But, for generating and applying/undoing legal moves, it is approximately 9 times faster than the baseline. When used as part of a program that strongly solves Chinese Checkers, a 2x improvement is seen on a small game with 63 billion states. Further experiments show that using the bitboard implementation without native `pext` and `pdep` support degrades performance to be slower than the original implementation for operations that rely heavily on these instructions.

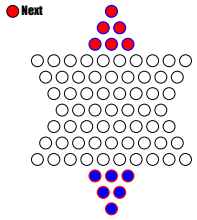

Fig. 1. 7x7 Chinese Checkers board

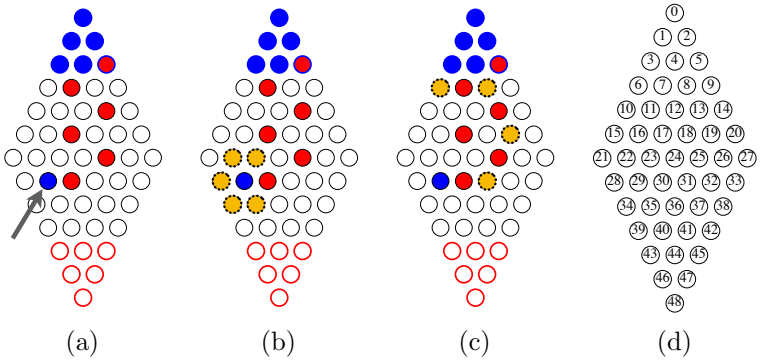

Fig. 2. Legal moves for the piece marked with an arrow in (a). Step moves are marked in yellow with a dashed circle in part (b), and jump moves are marked similarly in part (c).Numbering used for board locations is found in part (d).

2 Chinese Checkers Rules

Chinese Checkers is a game for 2-6 players that can be played on odd-sized boards, where we indicate the board size by the main playing area in a 2-player game. The most typical board is 9x9 with 10 pieces per player, but a 7x7 board with 6 pieces per player, as shown in Figure 1, is also common. The goal of the game is for players to move their pieces to the opposite side of the board from where they started. The winner is the first player to reach a goal state. Although pieces may jump over other pieces, but there are no captures.

Legal Actions: The four 'corners' on the sides the board are reserved as start and end locations for players in a game with more than two players. The rules do not allow players to place their pieces in these locations, although they are allowed to move through the corners as part of a longer action. There are two possible actions: moves that *step* to an adjacent empty location, and moves that *hop* or *jump* over an adjacent piece into a free location. Jumps can be chained together to move a piece far across the board in a single action.

Step Moves: Step moves are illustrated in Figure 2(a) and (b). Figure 2(a) shows the initial state where the blue player is to move; moves for the bottom blue piece, marked with an arrow, are considered. This piece is allowed to move

to five of its neighboring locations, because the sixth location is blocked by a red piece, as shown in Figure 2(b) by the yellow locations with a dashed border.
Jump moves: Jumping moves are allowed when a piece can move in a single direction, jumping over a neighboring piece, and landing in a free location on the board. From the state in Figure 2(a) the bottom blue piece is able to chain together up to four jumps. The locations the piece can move to are found in Figure 2(c) marked in yellow with a dashed border.

The choice of jumping moves requires a recursive search, and thus can be expensive. Bookkeeping is required to ensure that move generation doesn't continue to loop in circles generating the same moves repeatedly. The choice of stopping anywhere significantly increases the branching factor of the game.[3]
Win Conditions. The standard win condition of Chinese Checkers is for a player to fill their goal area with their own pieces. However, it has often been observed that in the two-player game a player can leave a single piece behind in their start area in order to prevent the opponent from filling their goal area. (While simultaneously insuring they do not win themselves.) To prevent this, we expand the win condition, only requiring that a player have at least one piece in the goal when it is filled with pieces. Thus if a single piece is left in the goal, it can just be surrounded to achieve a victory. Along with this rule, we do not allow a player to move backwards into their own goal to lose the game. This means that, for a player to win the game, they must make the last move. We have described further win conditions, illegal moves, and draws elsewhere [10], but these do not play a significant role in the implementation details described here.

3 Chinese Checkers Bitboard Representation

This paper describes a 7x7 bitboard implementation here that uses 64-bits to represent the board. In particular, the set bits in one 64-bit integer are used to represent the locations of a single player's pieces on the board. When allowing players to temporarily jump into unused corners, 61 bits are required: 49 bits for the main 7x7 diamond, and 3 bits for each of the four unused corners. Our 9x9 implementation uses two 64-bit states to represent the board, but is fundamentally the same as the 7x7 representation. For simplicity we only describe operations in terms of the central 7x7 board.

Operations on the board require a numbering system that map locations on the board to bits. There are many possible numberings that can be used on a board; our implementation just numbers states starting from 0 at the top of the board, as shown in Figure 2(d). While other mappings can be used, there are certain properties needed by the numbering to ensure that it can be used efficiently for some bit operations.

Our implementation has a separate board representation for the first and second player. Applying an `or` to these boards would result in a map of all

[3] At least 50 actions in 3-player Chinese Checkers [9], and 99 in single-player Chinese Checkers on the 9x9 board [12].

occupied locations. In the pseudo-code below we refer to the occupancy representation of all pieces in a state as `s.board`. When referring to a particular player we use `s.p1Board` and `s.p2Board`. In practice `s.board` is not stored, but is generated when needed. The state contains an additional variable `s.p1Move` which indicates whether player 1 is to move in the current state.

There are five basic operations we want to support in our implementation. These are: *GetMoves*, which returns the legal moves, *Apply/Undo Moves* which applies a single move to the game board, *GetWinner* which returns the winner of the game, if any, and *Rank* and *Unrank*, which compute a perfect hash from a state, and a state from a hash. We cover these operations in order from simplest to most complex.

3.1 Get Winner

The game is won when a player fills their goal area with pieces. We precompute a mask containing the goal area for each player. We use this mask to test (1) if the goal area is filled and (2) if a player has at least one piece in the goal. The code for testing if player 1 wins is as follows:

```
// p1 goal area filled
if ((((s.board)&p1Goal) == p1Goal) &&
  // p1 has at least 1 piece in goal
  (s.p1Board&p1Goal) != 0 &&
  // p2 to move
  s.p1Move == false)
  return 0; // p1 wins
```

The code for player 2 is analogous. The bitboard representation means that we can test all pieces in one operation instead of using a for loop to iterate through all of the locations in the goal.

3.2 Get, Apply, and Undo Moves

Because the number of legal moves in a state can be variable and large, but the number of pieces which can be moved is small, we describe an initial move representation that is more compact.

In particular, we have an array of 6 64-bit integers that represent the legal moves for each of the six pieces on the board for a player. The bits set in each 64-bit integer represent the locations that piece can legally move. For instance, looking at Figure 2(b) we can see that the piece in location 29 has five step moves and in Figure 2(c) we can see that it has four jump moves. Thus, the 6th integer (because this is the piece in the largest location) would have bits 22, 23, 28, 34, and 35 set for the step actions, and bits 6, 8, 19, and 31 set for each of the jump actions. Given this representation, we can discuss how we set these bits.

Step Moves: Getting legal step moves is straightforward using standard bit operations. For each location on the board we create a mask which contains

the neighboring states. A piece can only move to a neighboring state if it is unoccupied. Thus we can get the legal moves for a piece in location `pieceLoc` with the following operation:

```
legalMoves |= (~s.board)&neighborMask[pieceLoc]
```

In one operation this provides **all** steps that can be performed by a single piece, and these can be added to the legal moves for that piece using a bitwise `or`. This is significantly more efficient than a sequential implementation that loops over each possible action, checking to see if a location is free.

Jump Moves Unfortunately, using standard bitwise operations it isn't possible to simultaneously compute all jump actions. As a start, consider that we have two arrays representing the location jumped over, and the location jumped to, for a given start position. That is, for a piece in location 29 on the board, we would have a first array with values 22, 23, 30, and 35 indicating the locations to be jumped over, and a second array with values 15, 17, 31, and 40 indicating the locations where a piece would land after jumping. Note that the relative orderings of the state jumped over and the final location of the jump are maintained in the two arrays; these also happen to be sorted.

Assuming that these arrays are called `jumpOver` and `jumpTo`, the ith move can be tested with the following pseudo-code:

```
legalMoves |=(((s.board>>jumpOver[pieceLoc][i])&((~s.board)>>
    jumpTo[pieceLoc][i])&0x1)<<jumpTo[pieceLoc][i]
```

With this approach we could store an array of jumpOver and jumpTo locations for each location on the board. Each location has a maximum of six jump actions, but because some jump actions near the edge of the board are not legal, there will not always be six actions to consider.

Note that the array used for this version can get large, and thus it might be better to store the jumpOver and jumpTo locations more efficiently. In particular, we can have one 64-bit integer from all jumpOver locations and one 64-bit integer for all jumpTo locations. This works because the board numbering preserves the relative sorting of jumpOver and jumpTo locations. Thus, the ith bit in the jumpOver integer and the ith bit in the jumpTo location will both correspond to the same jumping action. Then the question becomes one of extracting out each of the bits and testing their values to see if the jump actions are legal. While this can be done by repeatedly counting trailing zeros and then clearing the low bit, the Intel BMI2 instruction set offers more efficient instructions.

Instead, we can use the `pdep` (parallel deposit) and `pext` (parallel extract) operations. `pext` uses a mask to extract bits from the input into the low-order bits of the output. `pdep` deposits these bits back into the locations indicated by the mask. These can be used to extract the locations that are jumped over and to to simultaneously test for legal jumps.

With this approach, just three lines of code are required:

```
jumpOver = pext(s.board, jumpOverMasks[pieceLoc]);
```

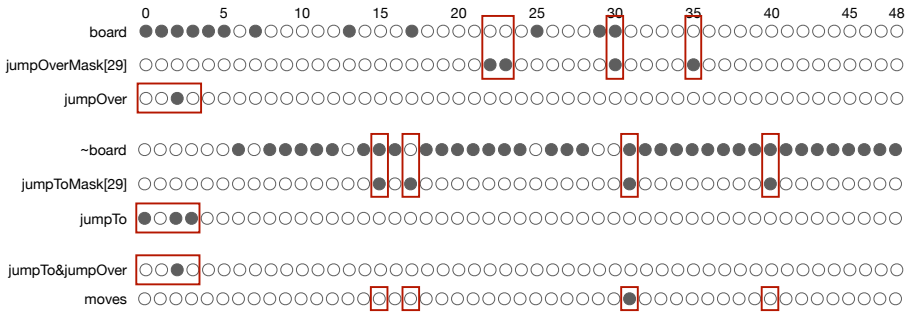

Fig. 3. Operations for computing jump moves

```
jumpTo = pext(~s.board, jumpToMasks[pieceLoc]);
moves = pdep(jumpTo&jumpOver, jumpToMasks[pieceLoc]);
```

The first line extracts the bits for the locations that are jumped over. These will be 1 if they are occupied. The second line extracts the locations that are being jumped to. Because the board is inverted, these will be 1 if they are unoccupied. By computing the bitwise `and` of these locations, we simultaneously test all directions to see if a jump move is possible. The last line deposits any legal moves back into the location that the piece is jumping to.

After jumps for a single pieces at a single location have been computed, they must be recursively computed for all new locations that a piece has arrived at in the previous calculation. This process continues until no new moves are discovered. The whole process is then repeated for all pieces.

To make this more concrete, we fully illustrate this process for the piece in location 29 in Figure 2 using Figure 3. There are four possible locations that this piece can jump over while remaining on the board. These bits are set in jumpOverMask[29]. The `pext` extracts the corresponding bits from the board and places them in the low bits of jumpOver. In order to perform a jump, there must be a piece in one of these locations to jump over. Corresponding to these locations are the locations that the piece would land after jumping, shown in jumpToMask[29]. These bits are extracted from the inverted board bits into the low bits of jumpTo. Doing a bitwise `and` on jumpTo and jumpOver results in one set bit for every possible jump. Using `pdep`, these bits are then deposited back into the original locations. The final result indicates that a piece in location 29 can jump to location 31.

This computes all jumps for a single piece in one step, instead of requiring a for loop over the 6 possible directions that a piece can move.

Apply and Undo Moves The results of finding all moves is a set of bits, one per location that a single piece can move. Each piece has its own set of legal move locations. This cannot be used directly to apply and undo moves. However, it is simple to extract the bits from this set and combine them with the bit for

the piece that is moving. Thus, a move action is represented by a 64-bit integer with two bits set, one for the initial piece location and one for the final location. The initial and final locations do not need to be distinguished.

Applying a `xor` to a player's board representation can be used to toggle both the bit for the current location of the piece as well as the location to which the piece is moving in one step. This process works regardless of whether a move results from a step or a jump. Undoing a move as part of a depth-first search is identical to applying a move.

3.3 Ranking and Unranking States

A key operation when solving a game, is ranking a state, or computing a perfect hash of a state. This allows data about state to be stored implicitly without explicitly describing the state – the location in memory implicitly identifies the state.

Although this calculation is not complex, efficient implementations are not widely documented in the literature. Thus, we provide a description of our approach here for reference. Our code has three ranking variants. These include ranking/unranking the pieces of the first player, ranking/unranking the pieces of the second player, and incrementing the location/rank of the second player's pieces relative to the first player's pieces. We focus on the ranking of the first player here, as this is the most common operation. The first player ranking does not rely on BMI2 instructions; the improvement found in experimental results comes from the improved ranking algorithm. The second player ranking is nearly identical to the first player ranking, except that the `pext` operation is used to get the board without the first player's pieces. The use of this instruction is important for overall efficiency.

Because the pieces on the board are not distinguished from each other, the ranking problem is related to the combinatorial ways the pieces can be placed on the board. Consider a board with 49 locations and 6 pieces, where the pieces are in locations $\ell_1 \cdots \ell_6$. There are $\binom{49}{6}$ total ways to place these pieces on the board. To compute the rank of an arbitrary state, we need to look at the gap between each pair of pieces to see how many possible ranks are skipped. That is, if the first piece is in location 0 on the board, we can decided to place the second piece anywhere from location 1 to location 44. If the second piece is placed in location 2, we can ask how many possible ranks of all pieces were skipped by that decision. Given the first piece is in location 0, and the second is in location 1, there are $\binom{47}{4}$ ways that the remaining pieces could be placed on the board. Thus, putting the second piece in location 2 increments the rank of the state by exactly $\binom{47}{4}$.

Generalizing this logic and starting with the first piece, when the first piece is put in location ℓ_1 there are

$$\sum_{i=\ell_1}^{49} \binom{i}{5} = \sum_{i=5}^{49} \binom{i}{5} - \sum_{i=5}^{\ell_1-1} \binom{i}{5} = \binom{49}{6} - \binom{\ell_1 - 1}{6}$$

possible rankings that have been skipped. Note that $\sum_{i=k}^{j}\binom{i}{k}$ is equivalent to computing the sum of the jth diagonal in Pascal's triangle, which is also equivalent to $\binom{j}{k+1}$. This justifies the last simplification of the formula.

In this case a state has a rank of 0 if no ranks are skipped; that is, all pieces are compactly placed at the top of the board in locations 0-5. Generalizing and counting all skips across all pieces we derive:

$$\left[\binom{49}{6}-\binom{\ell_1}{6}\right]+\left[\binom{\ell_1-1}{5}-\binom{\ell_2}{5}\right]+\left[\binom{\ell_2-1}{4}-\binom{\ell_3}{4}\right]\cdots\left[\binom{\ell_5-1}{1}-\binom{\ell_6}{1}\right]$$

By re-arranging terms, this can be re-written as:

$$\left[\binom{49}{6}\right]+\left[-\binom{\ell_1}{6}+\binom{\ell_1-1}{5}\right]+\left[-\binom{\ell_2}{5}+\binom{\ell_2-1}{4}\right]+\cdots\Bigg]-\binom{\ell_6}{1}$$

Since $-\binom{n-1}{k}=-\binom{n}{k}+\binom{n-1}{k-1}$, we can further simplify this to:

$$\binom{49}{6}-\binom{\ell_1-1}{5}-\binom{\ell_2-1}{4}-\binom{\ell_3-1}{3}-\binom{\ell_4-1}{2}-\binom{\ell_5-1}{1}-\binom{\ell_6}{1}$$

This provides a simple formula that can directly compute the rank of a state. When generalizing for a board with n locations and k pieces, the time required to compute the rank is $O(k)$, which contrasts with a similar ranking function described in previous work [4], which runs in time $O(n)$, and is the basis of the ranking we used for our previous work on solving single-agent Chinese Checkers variants [12, 8].

Note that our implementation pre-computes and caches the result of $\binom{n}{k}$ for all n and k that will be encountered in a given run. This cached result is returned from the `binom` function, which looks up the binomial coefficient for the given n and k. Other functions used in our code include `blsr`, which clears the lowest bit and `tzcnt`, which counts the trailing zeros.

```
int RankPlayer1(State s)
{
  int value = s.p1Board;
  result = kMaxRank;
  for (int x = 0; x < numPieces-1; x++)
  {
    result -= binom(boardSize-tzcnt(value)-1, numPieces-x+1)
    value = blsr(value); // clear low bit
  }
  result -= (boardSize-tzcnt(value));
  return result;
}
```

Unranking is the opposite process of ranking, but our code uses a simple $O(n)$ time algorithm, instead of $O(k)$ required for ranking. This could be made more

efficient, but unranking is used far less often than ranking, so it is less important to perform these optimizations in the code. The unranking algorithm loops over each possible location, testing whether the next piece should be placed in that location. When a valid location is found, a piece is placed, and the remaining pieces are iteratively placed in the same manner.

```
State UnrankPlayer1(int rank)
{
  State s;
  s.p1Board = 0;
  int nextLoc = boardSize;
  for (int x = numPieces; x > 1; x--)
  {
    do {
      nextLoc--;
    } while (binom(nextLoc, x) >= rank);

    s.p1Board |= 1<<(boardSize-1-nextLoc);
    rank -= binom(nextLoc, x);
  }
  s.p1Board |= 1<<(boardSize-1-(rank-1));
}
```

Note that a few details have been omitted from the code for simplicity of presentation; we are able to provide complete code upon request.

4 Experimental Results

We now evaluate our bitboard representations in comparison to the representation that was used for all of our previous published results on Chinese Checkers. The previous implementation is reasonably well optimized after 20 years of use. The original board is represented using three arrays: one containing the contents of the board, and two containing the pieces for each player. A DFS is used to find legal jump moves, with a bitboard used as a hash table for finding duplicates during the DFS. The move data structure contains which pieces is moving, as well as where it moves, which allows for efficient updating of the board. Ranking and unranking are performed using a variant of a previously published implementation [4]. Functions like Get Winner are implemented with for loops.

Experiments are run on a laptop with an 8-Core 2.3 GHz Intel Core i9, 32 GB of 2667 MHz DDR4 RAM, and MacOS Monterey. The code is in C++ and compiled with -O3. The Intel intrinsic headers are used for efficient bit operations. For comparison purposes a custom implementation of the `pext` and `pdep` functions was written using general bitwise operators. We call this implementation non-BMI2. We test each of the major game operations as follows, with timing results reported in microseconds.

For testing the GetWinner function, we make 1,000,000 GetWinner calls and report the total time averaged over 100 runs. For testing GetMoves, we get the

Table 1. Board with 49 locations and 3 pieces

Call	Original	BitBoard	non-BMI2
GetWinner	3,611	2,127	2,123
Get+Apply/Undo Succ	235,388	25,555	345,085
RankP1	5,455	3,095	3,097
RankP2	20,744	3,508	77,082
Unrank1	1,052	450	460
Unrank2	1,035	383	1,245
Increment	241	78	2,016

legal moves of a given state 1,000,000 times and then apply and undo all actions that were returned, reporting the total time. For the ranking test we rank the first 1,000,000 states for each player, reporting total time. For unranking and incremental ranking functions we fix one player and then unrank all possible states for the other player. The incremental unranking is only implemented for the second player.

The results in Table 1 show the results on the board with 49 locations and 3 pieces. For the GetWinner function the bitboard implementation is 1.7x faster than the old implementation. The non-BMI implementation does not have any significant overhead. For getting, applying, and undoing actions, the bitboard implementation is 9.2x faster than the old implementation with BMI2. Without the BMI2 instructions the new implementation is 1.5x slower. This is because of the extensive use of BMI2 functions. These trends continue across the ranking operations. There is little overhead for ranking player 1 without BMI2, but the same operations for player 2 are significantly more expensive.

The results on the board with 49 locations and 6 pieces, found in Table 2 follow the same trends as the smaller board, with the most significant gain being an 8.9x improvement in getting, applying, and undoing actions.

Table 2. Board with 49 locations and 6 pieces

Call	Original	BitBoard	non-BMI2
GetWinner	5,825	2,125	2,132
Get+Apply/Undo Succ	677,089	76,144	760,751
RankP1	7,957	5,171	5,868
RankP2	30,798	6,174	74,781
Unrank1	1,264,854	510,715	525,706
Unrank2	457,833	221,845	545,526
Increment	69,968	31,032	759,960

Table 3. Time in seconds to strongly solve games on board with 49 locations.

Pieces	Total States	Symmetric States	Original	BitBoard	Speedup
3	559,352,640	141,219,540	204.51	74.12	2.8
4	63,136,929,240	15,822,357,347	43,187.58	21,597.82	2.0

4.1 Solving Time

Finally, we take a sequential in-memory solver and use it to strongly solve boards with 49 locations and both 3 and 4 pieces. The solver uses both left/right symmetry on the board and symmetry between the players to reduce the total number of states that must be solved. The search uses retrograde analysis backwards from terminal states along with optimizations from previous work [11, 10]. In Table 3 we report the total solving time in seconds. The only difference in the code is the representation used for the game. Overall, the BitBoard representation leads to a 2x speedup in total solving time on the 7x7 board with 4 pieces, and a 2.8x speedup on the 7x7 board with 3 pieces. The reduction in speedup is likely due to memory overheads in the solver, as the entire game is stored in memory at the same time in this solver.

5 Conclusions

This paper has described an efficient bitboard representation for Chinese Checkers. This implementation provides a 9x speedup in the common operations of getting and applying successors, and a 2x speedup in strongly solving small Chinese Checkers boards. Future work will investigate faster bitboard methods for when BMI2 operations are not available, such as is the case on Apple's M1 processors.

References

1. Adel'son-Vel'skii, G.M., Arlazarov, V.L., Bitman, A.R., Zhivotovskii, A.A., Uskov, A.V.: Programming a computer to play chess. Russian Mathematical Surveys **25**(2), 221–262 (apr 1970)
2. Browne, C.: Bitboard methods for games. ICGA journal **37**(2), 67–84 (2014)
3. Carlini, S., Bergamaschi, S.: Arimaa: From rules to bitboard analysis. Knowledge Representation Thesis. University of Modena and Reggio Emilia (2008)
4. Edelkamp, S., Sulewski, D., Yücel, C.: Gpu exploration of two-player games with perfect hash functions. In: International Symposium on Combinatorial Search. vol. 1 (2010)
5. Frey, P.W.: An introduction to computer chess, pp. 54–81. Springer New York, New York, NY (1983)
6. Grimbergen, R.: Using bitboards for move generation in shogi. ICGA Journal **30**(1), 25–34 (2007)

7. Heinz, E.A.: How darkthought plays chess. ICGA Journal **20**(3), 166–176 (1997)
8. Hu, S., Sturtevant, N.R.: Direction-optimizing breadth-first search with external memory storage. International Joint Conference on Artificial Intelligence (IJCAI) pp. 1258–1264 (2019), https://webdocs.cs.ualberta.ca/ nathanst/papers/DEBFS.pdf
9. Sturtevant, N.R.: A comparison of algorithms for multi-player games. In: Computers and Games. pp. 108–122 (2002)
10. Sturtevant, N.R.: On strongly solving chinese checkers. In: Advances in Computer Games (ACG) (2019)
11. Sturtevant, N.R., Saffidine, A.: A study of forward versus backwards endgame solvers with results in chinese checkers. In: Computer Game Workshop at IJCAI. pp. 121–136 (2017), http://www.cs.ualberta.ca/ nathanst/papers/sturtevant2017ccsolve.pdf
12. Sturtevant, N., Rutherford, M.: Minimizing writes in parallel external memory search. International Joint Conference on Artificial Intelligence (IJCAI) pp. 666–673 (2013)