CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

GENERATING AN END GAME TABLEBASE FOR THE GAME OF BREAKTHROUGH USING QUASI-RETROGRADE ANALYSIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Andrew William Isaac

May 2016

The thesis of Andrew William Isaac is approved:

Diane L. Schwartz, Ph.D.	Date
John J. Noga, Ph.D.	Date
Richard J. Lorentz , Ph.D. , Chair	Date

California State University, Northridge

Dedication

Dedicated to my love, Morgan Aviva Alan, and my family for all their support.

Table of Contents

List of Figures

ABSTRACT

GENERATING AN END GAME TABLEBASE FOR THE GAME OF

BREAKTHROUGH USING QUASI-RETROGRADE ANALYSIS

By

Andrew William Isaac

Master of Science in Computer Science

End game tablebases are frequently used in chess to store game winning positions where few pieces remain on the board. Breakthrough usually has many more pieces than chess remaining on the board at game conclusion, making a standard style end game tablebase infeasible. However, due to the way the pieces in Breakthrough move, a table base entry for Breakthrough can instead be limited to a certain number of rows. By using quasi-retrograde analysis, forced wins can be found through reverse play from a winning position that are several moves away from game conclusion and may not be easily identifiable by a computer player. In theory, a computer player using Monte Carlo Tree Search can complete more simulations ending in a game outcome that can be known with certainty and therefore increase the likelihood of finding game winning moves if random play out can be terminated early via a query to an end game tablebase of proven forced wins.

Chapter 1

Introduction

1.1 The Game of Breakthrough

Breakthrough is a board game typically played on an eight column by eight row style board similar to chess and checkers. Developed by Dan Troyka, it was the winner of the 2001 8×8 Game Design Competition sponsored by About Board Games, Abstract Games Magazine, and Strategy Gaming Society. The game is notable for having surprisingly strategic play despite having simple rules that are easily learned. While Troyka initially designed the game for a seven row by seven column board, the game itself is trivially scalable to many other size boards. There is only one type of piece in Breakthrough that moves similarly to the way a pawn moves in chess. Initially the pieces are placed in the top and bottom two rows similar to the initial placement of pieces in chess, as shown in the left of Figure 1.1. [1]

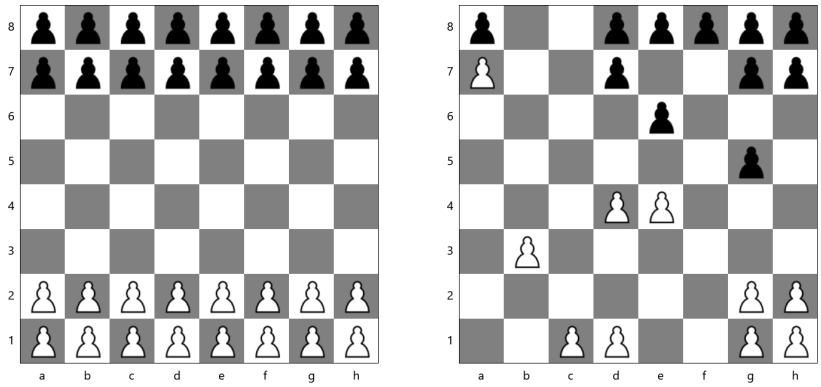


Figure 1.1: Initial starting positions and White to win in 1

To avoid confusion, the first player to move will play using white pieces and will be referred to as White and his or her opponent will play using black pieces and will be referred to as Black. While all strategy and moves will be discussed from White's perspective, the

[1] All chess piece images created by Colin M.L. Burnett and used with permission under CC-SA 3.0 license.

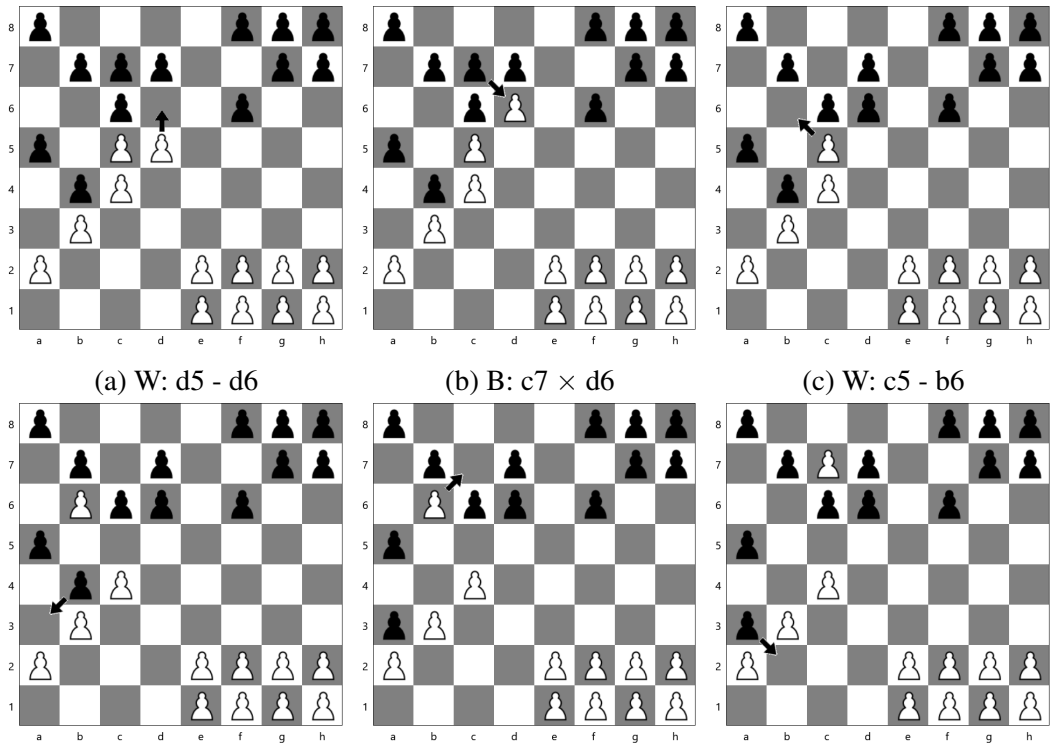
same moves and concepts apply to Black using a reversed perspective on the board.

When there are no obstructing pieces in front of a piece, that piece can move one row forward away from the original position and either one column to the left, straight forward remaining in the same column, or one column to the right. If the opponent has a piece one row forward and one column to the left or right of the player's piece, the player can capture the opponent's piece. White wins when either all of Black's pieces have been captured or White can move a piece to row 8. Black wins when either all of White's pieces have been captured or Black can move a piece to row 1. [6]

The strategic nature of the game becomes evident when the player successfully places his or her pieces so that the opponent must make a responding move either as a result of defensive play or in a $zugzwang$ situation that creates an opening that the player can then use to his or her advantage. This strategy is typically implemented by using some pieces as strategic sacrifices that force a chain of responding moves by the opponent that eventually will decide the game winner. For example, Black's failure to capture White's piece (c6 $\times$ d5) in Figure 1.2 shows White "breaking through" Black's defensive setup and leaving Black in a position where White's win cannot be stopped.

1.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic tree search algorithm that does random sampling of a decision based search space to find which decisions lead to the most favorable outcomes in a decision making process where a total sampling of the search space is not possible. For a simple game like 3 $\times$ 3 tic-tac-toe, it is possible to represent all possible games in the form of a search tree where each node represents a possible position and each child node represents a possible subsequent position. By assuming that a player will always choose the best move possible and his or her opponent will always choose the move that is in the opponent's best interest, it is possible to recursively iterate through the decision



(a) W: d5 - d6 (b) B: c7 × d6 (c) W: c5 - b6

(d) B: b4 - a3 (e) W: b6 - c7 (f) B: a3 - b2

Figure 1.2: A typical ”break through” play in Breakthrough

space for each move until a game outcome is reached. With these assumptions in mind, the best move for a player is the move that minimizes that maximum loss. [15, pp. 163-171] When both players follow this strategy perfectly, the outcome in tic-tac-toe will always be a draw since both players will always choose to minimize the maximum loss by taking a draw over a loss. When the search space becomes too large for it to be possible to know all possible game outcomes, a heuristic tree search like MCTS can provide a means in which a player can choose very good, if not the best moves, based on the moves that can be feasibly searched within a given time span. MCTS is named after Monte Carlo simulation, which is an approximation method used in a variety of applications to solve difficult problems by analyzing the output of an easier problem on a large random sampling that approximates the actual harder problem.

There are four stages involved in MCTS. The first stage is selection. A pure MCTS

approach would be to select each child node at random until a leaf of the tree is reached. Full MCTS uses a heuristic function called the upper confidence bound for trees (UCT) to select each child node. Once a leaf node is selected, the next stage is expansion. Unless the selected node represents a game concluding move, all possible remaining n moves left on the board for the opponent get a node added to the selected node, thereby expanding the tree by n nodes. The third stage is the random simulation stage. Unless an early play out termination evaluation function is being utilized, moves are simulated at random until game conclusion is reached. Finally, the game conclusion result is back-propagated up the tree from the expanded node to the root. This process is repeated as many times as possible within the given time span that a player has until his or her turn is complete. Once time is up, the move represented by the node with the highest score is selected as the best performing move and played on the board. [3]

1.3 End Game Tablebases

A forced win on a board is a state of a board game such that no matter how an opponent plays, if a player makes a correct move each turn, that player is certain to win. A classic example of a forced win is the concept of checkmate in chess. Once a player's king in chess is in such a position that no matter what a player does, his or her king will be captured on the following move, the game is declared over without the actual necessity of capturing the king. Building upon this concept, it is possible to find certain piece configurations on a board that are more than one move out from a certain or forced win. Finding these configurations in chess is often done using a process called retrograde analysis. The basic concept of retrograde analysis for finding forced wins in chess is to find all possible checkmate configurations that involve only a small number of pieces remaining on the board. Once the set of all possible checkmates that require these pieces (A_0) are found, all possible configurations such that a single move will create a board configuration contained within A_0 are added to the set of all board configurations that are one move away from checkmate (A_1).

Likewise, the set of mates in two moves (A_2) builds upon A_1. This process can be continued until all possible configurations using only those pieces up to n moves away from checkmate (A_n) are found. Each board configuration that is found to be n moves or less from checkmate can be stored using as a large unique integer. Without excluding impossible configurations, since there are twelve unique pieces in chess (counting both black and white) and there are sixty-four different positions on a chess board, there are 12^{64} different possible board configurations. Due to the size of this number and the limitations of current computers, it is impossible to iterate through all possible board configurations to determine which configurations are forced wins. By using a variety of optimizations to limit the total number of configurations that may need to be considered, a database representing various board configurations can be created that can be queried by the player during game play to see if a forced win exists when a small number of pieces remain the board. The database can be of several formats and can contain additional information such as the number of moves n that will be required and the set of moves that will be needed to reach checkmate in n depending on how the opponent chooses to play until checkmate is reached. [10, pp. 129-130]

In Breakthrough, an end game tablebase can be constructed in a similar fashion. From White's perspective, the simplest strategic plays involve only pieces located in the top three rows. If White has a piece on the top row, White has already won, and if White has a piece on the second most top row and it is White's turn to play, White's win is immediate. This makes any configuration of pieces on the board that have white pieces on either the seventh or eight row with White to move a won game for White. Therefore, assuming that White has a piece on the third row from the top and it is White's turn to play, how does White need to move in order to win the game? If White can move a piece from row 6 to row 8 such that that Black cannot capture White's piece, White has won the game. This means that Black must have defensive responses capable of capturing White's piece on row 7 to avoid a loss. Since Black must respond defensively or lose, any offensive strategic play that Black may

wish to pursue will need to be placed on hold for a turn. Since every time Black must make a defensive response at the expense of a offensive move, the progress towards Black's goal to win is stymied. It is therefore possible to consider White to have an effective forced win in n number of moves as long as Black doesn't have a better effective forced win in less than n moves.

1.4 Quasi-Retrograde Analysis

The traditional approach to retrograde analysis has been to select a mate that consists of a certain number of pieces required for a mate and then find all possible positions that all of the pieces could have moved from to create the mate. All the pieces in chess, with the exception of pawns, can move both forwards and backwards, so the entire board needs to be examined when using retrograde analysis to find possible forced wins. Since the pieces in Breakthrough only move in one direction away from each player's side of the board towards his or her opponent's side of the board, there will be pieces on the board that don't have any relevance to the game outcome. Due to this feature of Breakthrough, a variant of retrograde analysis can be implemented that finds forced wins involving only pieces in a certain number of rows instead examining the entire board for pieces that may be relevant. If a White piece can move to row 8, White can win immediately. Assuming reasonable play, White would never have more than one piece one move away from winning, since moving a second piece into a winning position instead of taking the win would unnecessarily prolong the game. So when White has a piece one move from row 8 and Black can not capture that piece, that is a forced win for White in one move involving just rows 7 and 8. Going further, any White piece on row 6 that can move to row 7 in such a way that no matter what move Black chooses as a response, White's piece on row 7 remains, will be a forced win in two moves. Unless Black has a piece on row 2 and can win immediately, if White has a piece on row 7, Black must attempt to capture that piece or Black will lose on the next turn. Therefore if Black is attempting to pursue an offensive strategy and White manages to move

a piece to row 7, Black must respond defensively at the cost of his or her offensive strategy and capture White's piece that has moved to row 7. Since the defensive response costs Black the chance to be one more move closer to winning and White only moved a piece that immediately was captured, essentially neither side is closer to winning in an effective count of the number of moves required to win. Any forced win for White involving pieces located in only the top three rows will always be an effective forced win in two moves. Once every forced win involving the top three rows is found, it is possible to repeat the process for the top four rows, i.e. if White can move a piece from row 5 to row 6 such that no matter how Black responds, White would have an effective win in two moves, then White has an effective forced win in three moves. Since it is possible that Black won't need to always respond defensively if White moves from row 5 to row 6, it is possible that White may need to make multiple moves before reaching a forced win in three moves. Each time Black is free to choose an offensive strategy, i.e. not responding to White's move won't necessarily leave White closer to winning, it means that White is effectively one more move away from winning. For example, Figure 1.3 shows White pursuing an effective forced win in three moves that actually requires seven plies since the first two moves do not require a defensive response by Black.

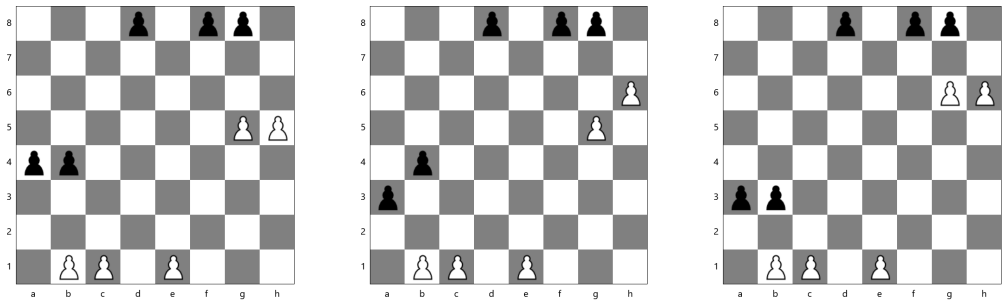


Figure 1.3: White pursues an effective win-in-3 requiring 7 plies

1.5 Constructing an End Game Tablebase for Breakthrough

The goal of this Master's project is to demonstrate the construction of an end game tablebase for a Monte Carlo Tree Search based computer Breakthrough player. By using

quasi-retrograde analysis, all forced wins that exist on the board that use only the pieces within a certain number of rows from the goal can be found. Each forced win that is found can be encoded as an integer using Gödel numbering and stored within a searchable data structure so that the computer player can quickly consult the tablebase during game play. In theory, if the computer player can consult the tablebase faster than the time it takes to simulate all possible outcomes from a given node and know if a board state is a forced win, the computer player can complete a greater number of simulations prior to move selection and therefore have a greater chance of selecting a better move. Since memory size is a significant limiting factor when creating tablebases, a proof of concept tablebase for a 6 x 6 board was first attempted to determine the feasibility of this project. Initial results looked to be promising enough that it seems that it might be possible that a strong MCTS based Breakthrough player enabled with a tablebase can correctly determine a game winner for a 6×6 board at the very onset of the game. However, a tablebase containing all forced wins involving the topmost 4 rows proved to be insufficient for this task, despite showing significant improvement in actual game play. A tablebase containing all forced wins involving the topmost 5 rows of the board was found to be infeasible due to memory size limitations and the length of time that it would take to iterate and test all possible piece configurations within 5 rows. Unfortunately, the same limitations prevented a complete construction of a 4 row tablebase for an 8×8 Breakthrough board, but with sufficient memory resources, it is believed that a similar significant performance improvement could be achieved in 8×8 Breakthrough.

Chapter 2

Monte Carlo Tree Search

2.1 Introduction

Monte Carlo Tree Search (MCTS) is a heuristic tree search algorithm that attempts to use random sampling of a decision-based search space to find near optimal results. Named after Monte Carlo methods used in a wide variety of scientific and engineering applications that use random sampling to achieve a heuristic result that otherwise can not be easily calculated, MCTS has been implemented in a large number of highly successful game playing programs. One the most recent successes of MCTS has been the remarkable improvement of computers playing in go. Until recently, the best go playing programs could only play at the best amateur levels. However by using a combination of machine learning and MCTS, Google's Deepmind AlphaGo project won 4-1 against the best rated human player Lee Sedol in March 2016. [17] Game playing programs using MCTS have also shown remarkable success in other games including Havannah and Amazons [13] as well as non-deterministic games like poker. [14]

One of the first known accounts of using a Monte Carlo method was by Stanislaw Ulam in 1946 when he was working on the hydrogen bomb project at Los Alamos. Because the actual calculations of the differential equations for calculating neutron diffusion were computationally difficult at the time, Ulam proposed using random sampling to find results with statistical analysis that otherwise could not be easily computed. [5] Widely applied in many fields including physics, biology, and graphics, Monte Carlo methods were used for some game playing tree search applications prior to 2006 but not given widespread attention until Remi Coulom coined the term Monte Carlo Tree Search and demonstrated how the MCTS based go playing program CrazyStone had a significant performance advantage over the best go playing programs at the time. [4] Another computer go player called MoGo was

developed around the same time and also used MCTS to become the first ranked program on the 9×9 Computer Go Server in August 2006. [20]

2.2 Tree Search Space

For any zero sum game with perfect information like tic-tac-toe, chess, go, or Breakthrough, at any point in the game there will always be a set of optimal moves and a set of sub-optimal moves that the player must choose from. The difficulty of these types of games and what makes them fun to play is that it is not always immediately evident which valid moves are optimal and which are sub-optimal. The optimality of a move only becomes evident once subsequent moves take place and the game conclusion that results from a series of optimal or suboptimal moves that a player chose has been reached. Starting from the beginning of the game, each board state can be represented as a node in a tree. When a player decides on a move, the board state changes to a new state. Each new state can be represented as a node in a tree where each child node is a possible new board state that a player may choose based on the legal moves available for the current state of the board. A tree holding all possible board states can therefore be theoretically searched from the root, which represents the initial starting positions on a board, to an outcome that is most optimal for a particular player. Since a player would much rather win (or draw in a game where draws are possible) than lose, an algorithm that always seeks to minimize the maximum loss (minimax) can theoretically search a tree for all possible outcomes, find the optimal move by assuming that the opponent will also always choose the optimal move as well, and provide the player with optimal move choices for every state of the game. If the opponent does not play optimally, then the advantage goes to the player following a minimax algorithm and that player will win (or reach a draw in a game where draws are possible). For a simple game like 3×3 tic-tac-toe, it is possible to create a tree that holds all possible board states. (A full tree for 3×3 tic-tac-toe has less than 9! nodes when accounting for games that would be the identical if the board were to be rotated some number of times as

well as games that finish early when a player gets 3-in-a-row with empty squares remaining.) For more complicated games like chess or go, the total move sequences, which can be represented as paths in a tree from the root to a leaf node representing a game conclusion, are much larger. Early estimates for the potential number of chess games calculated by Claude Shannon gave 10^{120} possible games [16] while more recent calculations by Victor Allis give a smaller upper bound of 10^{50} games [1, p. 171]. The game tree complexity for go is much larger, with estimates for the total number of go games to be between $10^{10^{48}}$ and $10^{10^{171}}$ games. [19] Such large game trees can not possibly be contained within current computational data storage, so an alternative method of searching a game tree like MCTS that does not need to consider all nodes to still provide good results is necessary. In fact, it has been proven that MCTS with unlimited computational resources will converge to a minimax algorithm that finds an optimal solution. [9]

2.3 The Four Stages of MCTS

The nodes in the tree built by MCTS need to keep track of at least three values in order to determine the best performing nodes within the tree and therefore the best move that the player should select: the number of times a node has been tried, the number of times that when a node was selected it led to a win, and the current player's turn the node represents. The MCTS algorithm iterates through the four stages shown in Figure 2.1 until either a requisite number of simulations have been completed or a time to turn limit is reached and the best move found by the algorithm is selected for play. [1]

2.3.1 Selection

The first stage of MCTS is selecting a node within the search tree to further expand. Selection starts at the root node of the tree and then recursively chooses one of the current node's children using a heuristic formula that chooses the best child node for selection by

[1] MCTS Stages image created by Wikipedia user Mciura and used with permission under CC-SA 3.0 license.

weighing the value of a node based upon its previous performance and its potential future performance. This recursion continues until a leaf node in the tree is reached.

2.3.2 Expansion

Once a leaf node is reached, the process of expanding the node by giving it child nodes begins. Based upon the previous player's turn, as indicated by the selected leaf node, all possible moves by the next player are added as nodes to the selected leaf node. Then one of these child nodes are selected to begin the simulation stage.

2.3.3 Simulation

Once a node has been expanded, the process of determining the performance of a node begins. Since not all possible move sequences can be examined, a move sequence from the board state represented by the expanded node to a possible game conclusion is chosen at random. In some cases, the simulation can be terminated early if a strong heuristic function can be employed that will reliably indicate a game winner even if the game still has not been simulated to completion. This type of simulation is known as early playout termination. [11]

2.3.4 Backpropagation

Once the game has been simulated to completion or reached early playout termination, the task of recording the result begins. Backtracking up the tree from game concluding node to root, each node that was visited gets an additional try added to its count. If the game concluded favorably for the current player, its win count is also incremented. These four stages are repeated for a maximum number of iterations or as many times as the turn time limit will allow, increasing the size of the tree during each iteration during the expansion stage. Once the time limit or maximum number of iterations have been reached, the child node of the root that showed the best performance represents the move that should be selected by the player. [3]

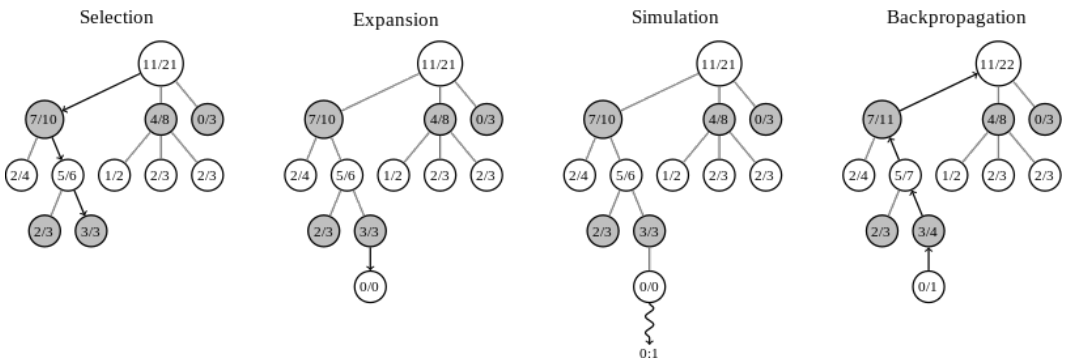


Figure 2.1: The four stages of MCTS

2.4 Upper Confidence Bound for Trees

A pure Monte Carlo Tree Search would use randomly selected nodes for expansion. The problem with such an approach is that after a sufficient number of trials, it becomes evident that some nodes tend to perform better than others. A random selection would continue to choose mediocre nodes with equal likelihood as good nodes and nodes that have been tried frequently with equal likelihood as nodes that have been tried infrequently. The need to exploit nodes that tend to perform well with the need to explore nodes whose performance is yet unknown requires a careful balance that is known as a bandit problem. A bandit problem is often visualized as a scenario in which a gambler has to choose which levers to pull on k gambling machines. While initially the gambler doesn't know which machines will yield good results, after pulling enough levers, he or she can begin to learn which machines tend to yield good results and which machines need to be tried more to determine their potential payoff. Since the goal of MCTS is to find the nodes that lead to the best results, a selection formula that weighs the potential payoff of unknown nodes versus the known payoff of previously tried high performing nodes is needed. For different applications of MCTS, the formula to balance exploitation with exploration will vary. This formula, coined an upper confidence for trees (UCT) by Kocsis and Szepesávri, will adjust the weight of nodes in the tree based upon how often they have been chosen in the past as well as how often they led to a positive outcome. A possible implementation is shown in

formula 2.1, where w_i is the number of times trying node i has led to a win and n_i is the number of times node i has been tried, t is the number of times the parent of n_i has been selected, c is some constant that leads to an optimal balance between the exploration and exploitation sides of the formula (found through experimentation), and k is the number of children belonging to the parent node. [8]

$$\max_{1 \leq i \leq k} \left(\frac{w_i}{n_i} + c \times \sqrt{\frac{\ln t}{n_i}} \right) \tag{2.1}$$

2.5 Wanderer: A MCTS Based Breakthrough Player

There are two locations where a MCTS based Breakthrough player can query a tablebase of forced wins. The first logical location is when a node is initially expanded. If the node representing a possible move leads to a piece configuration that exists as a forced win within a tablebase, there is no need to conduct a random play out. Instead, the node can be immediately scored and back propagated up to the root as a win. The second logical location to query a tablebase is during the simulation stage when the game is running a sequence of randomly selected moves to game completion. If the board reaches a state that exists as a forced win in the tablebase, the random play out can terminate early. Theoretically, if the time it takes to query the tablebase is less than the time it takes to complete the random playout of a game from an expanded node, the MCTS algorithm can complete more simulations per turn. Such a player could potentially find a better move than a player that does not terminate play out early with a tablebase that contains a comprehensive collection of end game forced wins. Also, the tablebase value will be more accurate, since it would not be an approximation showing a likely win, but rather a known forced win. In order to compare the effect that a tablebase may have on an MCTS player, a strong MCTS based computer player known as Wanderer was altered to query a tablebase constructed using a quasi-retrograde analysis process described in Chapter 4. Developed by Richard

Lorentz and Therese Horey, the original version of Wanderer uses an evaluation function to terminate play out early during the simulation stage. [12] Wanderer was altered so that the tablebase would be queried just prior to when the evaluation function is called. If the tablebase shows the current player has a forced win, the play out is terminated early and the win is back-propagated up the tree to the node. Otherwise, the evaluation function is run just like in the original version. Comparisons between two versions of Wanderer, one of which does query a tablebase during the random playout, and the other which does not, reveals that querying a tablebase during the random playout of a simulation can in fact improve the quality of a MCTS based Breakthrough player. More detail about how the end game tablebase affected Wanderer's performance can be found in Chapters 5 and 6.

Chapter 3

Chess End Game Tablebases

3.1 Introduction

Richard Bellman proposed in 1965 the idea of using retrograde analysis to create a database of optimal moves in chess to solve certain endgame situations. At the time, computational resources were not sufficient to actually create a tablebase of forced wins involving just three pieces. [2] However, advances in computational resources have since led to the creation of seven piece tablebases such as the Lomonosov tablebases generated with the respectively named supercomputer. These tablebases are made by considering all possible checkmates of some combination of n pieces and then considering all possible moves that lead to one of the known checkmates. Once all of these entries that are one move away from checkmate are found, all the possible moves that lead to one of those board positions are found next. This process is repeated until no further configurations are found in order to create a tablebase that contains all forced wins that involve up to n pieces on the board.

3.2 Finding All Checkmates Involving n Pieces

Since any board with only two kings remaining is a draw in chess, a checkmate will always require at least three pieces, a king and some other piece for White and a king for Black. An examination of the chess board reveals that there are only ten unique places on a chess board that Black's king can be placed. All other positions can be considered to be equivalent to one of the unique positions if the board is either rotated or symmetrically mirrored. Since White's king can not be placed next to Black's king, there are at most 60 other positions White's king can be placed. Once both kings are placed, the third piece can be placed in 62 other possible positions. Thus, there are only about 40,000 possible board configurations (some configurations will be impossible) that need to be evaluated as

possible checkmates involving the two kings and some other piece which is not a pawn. Pawns increase the complexity since they can only move in one direction while the other pieces can move in any direction that allows for a valid move. Other considerations that need to be taken into account in order for the position to be evaluated as a checkmate are which player is to move next and which player owns the third piece. Some situations will arise where the checkmate has already been identified except that the player color is simply reversed. These can be ignored since the reverse situation can always be queried instead of adding entries in the tablebase that are identical except for a difference in player color. Each configuration is examined to see if a player is in checkmate and all situations where checkmate exists get added to the tablebase. Once all three piece checkmates are identified and added to the tablebase, the same process is repeated for all four piece checkmates, and then all five, etc. [10, pp. 129-130]

3.3 Using Retrograde Analysis to Find Checkmate in m Moves

Once all checkmates involving n pieces have been found, then all possible locations on the board that each piece of each checkmate added to the tablebase could have moved from are examined to see which moves may have led to the checkmate. For instance, Figure 3.1 shows how a possible move could take place that leads to Black checkmate by first finding a White ply that puts Black into checkmate. After every possible ply by White is found that could lead to a checkmate by using an un-move generator, every possible ply by Black is examined using the un-move generator with White's following ply un-moved to see which combination of Black and White plies could have led to Black's checkmate. When a piece configuration is found such that no matter how Black plays, White can make a move such that Black will always be in checkmate, then an entry can be added to the tablebase that gives White a mate-in-1 move. Once this process has been completed for all possible moves and all mates-in-1 have been found, the process is repeated to find all possible moves that lead to each mate-in-1 in order to find all mates in 2, 3, and so forth up

to $m - 1$, where no more mates-in-m using n pieces can be found. [18]

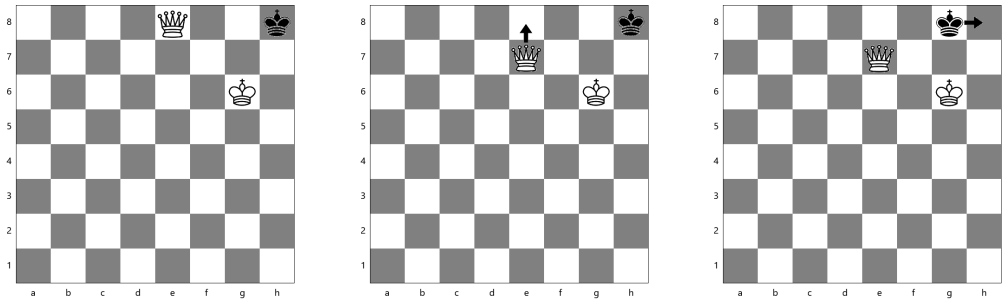


Figure 3.1: Finding a mate-in-1 move using retrograde analysis

3.4 Querying End Game Tablebases During Game Play

An obvious advantage of using an end game tablebase for a computer player is that it saves the programmer the difficult task of developing an algorithm for strong end game play. Instead, once the computer realizes that it has reached the beginning of the end of a game, i.e. the piece count has dropped sufficient low, it can instead query the end game tablebase to determine which moves will lead to mate. The biggest disadvantage is that end game tablebases are often very large and take away memory resources that a game playing algorithm may need to run enough simulations to find the best possible solution within the available time and memory limitations. Additionally in cases where each turn is limited by a set amount of time, if the tablebase is too large to be solely resident in random access memory, disk seek time can have a substantial negative impact on player performance. To address this performance issue, compact six piece tablebases have been created that are small enough to fit on solid state drives to eliminate the delay caused by disk seek time. [1] The most widely used tablebases are the 3 to 6 piece tablebases created by Eugene Nalimov. At 1.2 terabytes for the 6 piece tablebase, this tablebase can not

[1] More information about Syzygy tablebases small enough to fit on a solid state drive can be found at: https://github.com/syzygy1/tb

be cheaply stored on a solid state memory drive, but does completely solve all 6 piece chess positions. [2] Seven piece tablebases have also been created using the University of Moscow's Lomonosov super computer. This tablebase contains over 5 x 10^{14} entries and requires over 100 terabytes of storage space when compressed. [3] Given the storage size of such tablebases, the performance advantage of having an end game oracle comes at a certain cost that may not be the most beneficial use of memory resources.

[2]The 6 man Nalimov tablebase can be queried on-line at: http://www.k4it.de/?topic=egtb&lang=en
[3]More information about the Lomonosov tablebase can be found at: http://tb7.chessok.com/

Chapter 4

Quasi-Retrograde Analysis

4.1 Introduction

Since the pieces in Breakthrough can only move towards the goal row, a piece can no longer have any effect on pieces it has passed. For example, once White moves a piece from row 3 to row 4, that piece cannot block or capture any Black piece on rows 1 through 4 and would only be defensively effective against pieces located on rows 5 through 8. Due to this aspect of Breakthrough, it is possible to find all possible forced wins that only involve pieces located within the top n rows of the board and know with certainty that unless the opponent has a better forced win within the bottom n rows, that the player will win without needing to analyze the positions of any pieces not located within the top or bottom n rows. Since it is not known which column, $m-1$, m, $m+1$, a particular piece may have come from when it moved from row $n-1$ to row n, nor what other pieces might have been located on row $n-1$, all possible White piece permutations on row $n-1$ need to be considered when trying to find wins using retrograde analysis. By iterating through each possible piece configuration involving just n rows in a specific order, it is possible to find all possible forced wins that involve pieces located within the top or bottom n rows. Since a tablebase of forced wins constructed for White can be simply reversed to create a tablebase for Black, the process of creating the tablebase will be described from White's perspective with White to move on a 8×8 board.

4.2 Iterating Through Piece Configurations

Any White piece on row 8 is a win for White, so any piece configurations with White pieces on row 8 do not need to be considered nor included in the tablebase. If White has a piece on row 7 and it is White's turn to move (which it will always be since it is assumed

that the tablebase will only be queried when it is the current player's turn to move), then White will win in 1 move. Therefore board configurations involving just rows 7 and 8 will not need to be considered or included in the tablebase either. When White queries the tablebase during game play, checking for any white piece on row 7 will suffice for the detection of any win that can be completed in 1 move. Therefore piece configurations in the three topmost rows will need to be the first configurations that get analyzed in order to determine which configurations lead to a forced win for White. The simplest forced win involving three rows would of course be a configuration where Black has no pieces in the top two rows and White has a single piece on row 6, as shown in Figure 4.1 (left). Since it is irrelevant to White's forced win if Black has a piece on row 6 (since such a piece can only move to row 5), all combinations of Black pieces starting with row 7 and then proceeding to row 8 with White having a single piece on row 6 are next considered. If White has a single piece on row 6 and Black has some combination involving pieces on either row 7 or row 8 or both that White can either capture or will not block White's piece in any way, this would also be a forced win for White, as shown in Figure 4.1 (right). Just as White having a piece on row 7 is a win-in-1, a situation where White has a piece on row 6 that can not be captured by Black is a win-in-2. By ordering the pieces such that White's pieces increment in quantity from higher numbered rows to lower numbered rows and Black's pieces build in a similar fashion from lower numbered rows to higher numbered rows, all possible forced wins can be found that exist for the top three rows, since either White eventually is able to move a piece to row 7 that Black cannot capture, giving White a forced win-in-1, or Black can capture it, but the resulting configuration will be one that has been previously found to be a forced win since there are now less pieces on the board. The pieces can be incremented on each row similar to the way binary numbers increment. With White starting on row 6, initially a single white piece on column a is placed. Once all configurations of Black pieces are tested with the White piece located at position $a6$, the White piece at $a6$ is removed and a White piece is placed at $b6$ and once again all configurations of Black

pieces are tested. Then two white pieces are placed at positions $a6$ and $b6$ and all possible black piece configurations are tested. This is repeated until every possible combination of up to a certain piece limit have been tried starting with row 6 and moving downwards. Each configuration of Black pieces is similar incremented, but skipping over any position that may be currently occupied by a White piece.

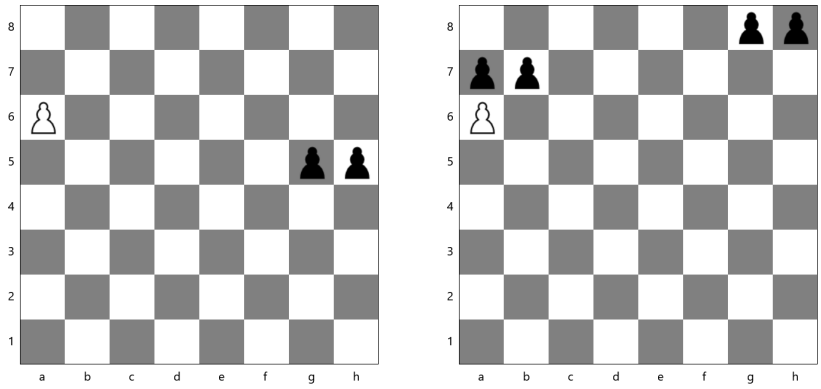


Figure 4.1: Simple wins-In-2

4.3 Finding a Forced Win

A piece configuration can be considered to be a forced win in some n number of moves for White if White can make a move such that, for every move Black can make in response, the resulting piece configuration was previously found and added to the tablebase. There can however be multiple forced wins for White for a given piece configuration. Since the tablebase should contain the most effective forced win, it is also necessary to test each of White's possible moves to see which moves do lead to a forced win and which of those moves would be the most effective. Once the most effective forced win is found, the board state can be encoded into an integer along with the best recommended move for White and the effective number of moves required, as described in Section 4.6.

4.4 Finding the Effectiveness of a Forced Win

In order to tell if White has a forced win that is better than a forced win Black might have, there needs to be automated process that gauges the effectiveness of a particular forced win. Since there are situations in which Black must respond defensively to one of White's moves or otherwise lose the game, the total move count of a forced win doesn't necessarily indicate the effectiveness of a forced win. For example, in Figure 4.2, White's forced win takes 7 total plies $(E6-E7, d8\times e7, D6\times E7, f8\times e7, F6\times E7, c3-c2, E7-D8)$ but effectively it is only 3 plies because Black must respond defensively twice at the cost of pursuing his or her own unobstructed forced win using the piece located at $c3$. The effectiveness of a forced win is calculated by checking if Black's failure to make some move results in a more effective forced win for White. Black's optimal move will always be to pursue an offensive move over a defensive move unless the defensive move is necessary to avoid a loss.

In order to find a forced win, all possible White moves for a given board state are tried. For each White move, all possible resulting Black moves are tried. If the resulting board state for each Black move tried is a forced win that has been previously found and added to the tablebase, then White's move is known to be also a forced win. Since the piece configuration is a forced win for White, it doesn't matter which move Black chooses, Black will still lose. It is useful however to identify which move by Black will prolong the game the longest and assume that is the move Black will choose. This gives an accurate count of the total number of moves remaining until White wins, i.e. a forced win-in-n may take less than n moves if Black just lets White move a piece towards the goal without trying to block or capture it. If Black does not make a move that prolongs the game for as long as possible, White can then take advantage of Black's move and can achieve a more effective forced win. If it doesn't matter whether Black tries to prolong his or her inevitable loss or not, i.e. there is no way to block or capture any pieces White will use to win, then Black's

best choice would be to pursue an offensive strategy and try to win first. In this case, White's effective forced win will be a forced win-in-$n+1$ where n is the number of moves needed to complete the effective forced win that was previously added to the tablebase. Essentially, when Black's best course of action is an ultimately futile effort to stop White from winning, then the number of actual moves required for White to win doesn't increase beyond the previous total number of moves found. When Black's best course of action is to try to win first, then for each White move, Black can get one move closer to winning also, so White's effective forced win takes an additional move. This metric allows a player to determine when both White and Black have forced wins in the tablebase if the player or the opponent will win first.

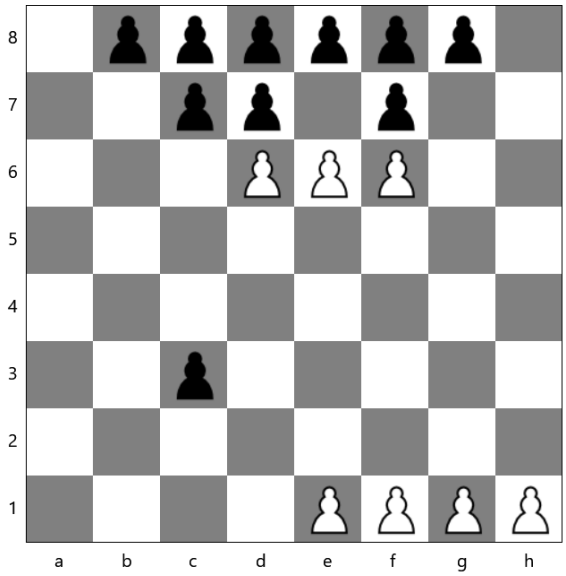


Figure 4.2: An effective win-in-2 for White requiring 7 plies

4.5 Symmetric Optimization

Once a particular board configuration has been found to be a forced win, the mirror image along the vertical axis can also be immediately added to the tablebase as a forced win if the player makes a similarly reversed move. When the iteration encounters a board configuration that has already been added to the tablebase, it can skip it instead of testing

the configuration, thereby nearly halving the run time required to iterate through all board configurations, as shown in Figure 4.3.

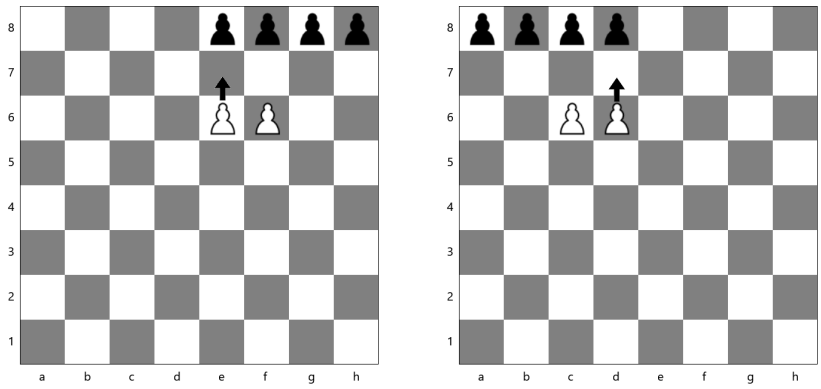


Figure 4.3: A forced win and its mirror image

4.6 The Forced Win as an Integer

In order for a board state to be contained within a tablebase that can be searched efficiently, the state of the board first needs to be converted to an integer representing the board state. First each position i on the board needs to be indexed, starting with the top left position (a8) and going left to right and then from the top row down. Then the piece configuration can be encoded by the formula in Equation 4.1 where k is the product of the number of rows and columns that the complete tablebase will include. Additionally, the best move White should make and the effective number of moves required to win are needed to correctly identify whether White will win. By indexing each column 0 through 7 from left-to-right and each row 0 through 7 from bottom-to-top, each move on the board can also be encoded as an integer, as shown in Equation 4.2. By adding these values together along with the effective move count, each forced win that is found will have an integer value in the tablebase that, when decoded during game play, will return the best move and the total number of moves of the effective forced win.

$$\sum_{i=0}^{k-1} \begin{cases} 3^{k-i-1} \times 2, & \text{if i contains White} \\ 3^{k-i-1}, & \text{if i contains Black} \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

$$toRow + toCol \times 8^1 + fromRow \times 8^2 + fromCol \times 8^3 + effectiveMoveCount \times 8^4 \tag{4.2}$$

4.7 The Tablebase Data Structure

Two significant limiting factors when deciding to use an end game tablebase are the size of the tablebase and the time it takes to query the tablebase. When the tablebase for a Breakthrough 6 $\times$ 6 board was initially being created, the first choice was to use the C++ 11 standard library hash map implementation since hash maps have constant search time. However, since the C++ 11 standard library implementation of the hash map has significant pointer overhead, there were insufficient memory resources available to generate the 5 row by 6 column tablebase discussed in Chapter 5 using the standard library hash map. One solution found was to use Google's open source C++ B-Tree map library instead. This significantly reduced memory usage and allowed a non-comprehensive 5 row by 6 column tablebase to be created that uses just over 4.7 gigabytes of memory. Although the B-Tree implementation requires logarithmic time to search, it was discovered to be actually more efficient than the standard library hash map implementation. Using a tablebase with 87 million entries, on average Wanderer playing on a 6 $\times$ 6 board with 5 second turns was able to complete 256,535 simulations per turn using the standard library hashmap and 330,276 simulations per turn using the B-Tree. [1]

[1]More information about Google's open source C++ B-Tree library can be found at: http://google-opensource.blogspot.com/2013/01/c-containers-that-save-memory-and-time.html

4.8 Generating the Tablebase by Rows

Once all forced wins involving the top three rows have been found, the same process can be repeated for the top four rows. Each forced win found will either be the result of finding a move by White such that no matter which move Black makes, it results in a piece configuration that was previously found to be a forced win involving pieces in either the top four rows or the top three rows. This same process could theoretically be repeated with sufficient time and memory resources until the entire 8×8 board is solved, but a tablebase for just the top four rows with unlimited piece counts could not be created within a feasible amount of time nor with a machine with less than several hundred gigabytes of random access memory. See Chapter 6 for details regarding how a four row end game tablebase for an 8×8 board with a limited number of pieces was constructed.

Chapter 5

Results Achieved in 6 × 6 Breakthrough

5.1 Introduction

The first tablebase created was for a 5 × 5 board in order to compare results with a Breakthrough oracle for 5 × 5 boards created by Jan Haugland. [1] A 5 × 5 board is sufficiently small that by using quasi-retrograde analysis, every possible piece configuration can be solved, including the initial starting configuration, which Black wins with optimal play. After a series of tests comparing results between the 5 × 5 tablebase constructed using quasi-retrograde analysis and Haugland's oracle and finding no differences in outcome, the process of creating an end game tablebase for a 6 × 6 board began. With over 13 trillion different piece configurations on a 6 × 6 board (not including configurations where White pieces are in the top two rows or Black pieces in the bottom row), it becomes evident that some sort of piece limitation is required, like the piece limits used in chess end game tablebases.

5.2 Solving 6 × 6 Breakthrough

During tests with a limited end game tablebase created for 6 × 6 that covers every row except the pieces in row 1 for White and row 6 for Black, it was hypothesized that Wanderer could potentially be used to solve 6 × 6 Breakthrough through the aid of the end game tablebase. While Wanderer with sufficient time and memory resources could solve Breakthrough 6 × 6 without the tablebase, since MCTS eventually converges to minimax, the tablebase could potentially save time and tree node allocation space by allowing Wanderer to immediately eliminate nodes that lead to piece configurations that are known losses. By using an end game tablebase with limited piece counts, Wanderer was able to solve piece

[1] Haugland's 5 × 5 Breakthrough oracle can be found at: http://www.neutreeko.net/neutreeko.htm

configurations like the one shown in Figure 5.1 that are two moves from the initial piece configuration. A 5 row tablebase that has a higher piece limit than the tablebases that have been created thus far could potentially lead to solving Breakthrough 6 $\times$ 6 for White, but such a tablebase large enough to fully solve Breakthrough 6 $\times$ 6 has yet to be created. [7]

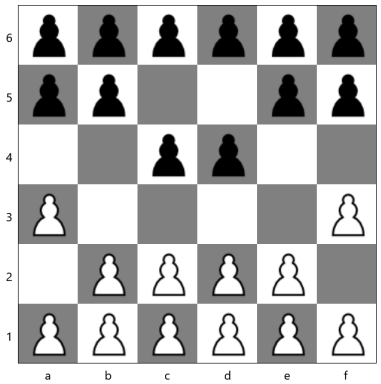


Figure 5.1: White wins after 2 moves from initial configuration

5.3 Initial Tablebase Results

Initial results when a 5 row tablebase enabled version of 6 $\times$ 6 Wanderer was played against the original non-tablebase enabled version were promising. (These tests used a version of the tablebase that excluded all forced wins that showed Black having the potential to have a better forced win for White.) The 5 row tablebase only has entries for forced wins involving pieces in rows 2 through 6, so it can't be known if White has a piece on row 1 that could capture Black's piece located at $a2$ in Figure 5.2. As a result, such a configuration would be a forced win for Black if White can not capture Black's piece. If White can capture Black's piece at $a2$, then White's forced win would be dependent upon Black's responding move. If every move by Black still results in a forced win for White, then that would be added to the 6 row tablebase. This tablebase version however was not comprehensive and was insufficient for solving 6 $\times$ 6 Breakthrough. In terms of gameplay, when a version of Wanderer using this tablebase version performed noticeably better than the non-tablebase version, as shown in Figure 5.3.

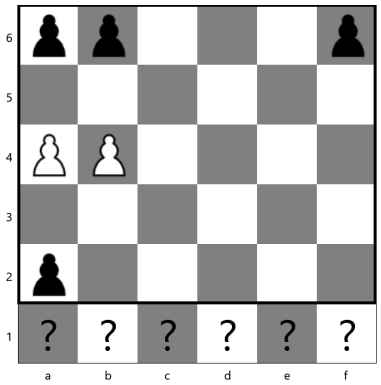


Figure 5.2: A 5 row configuration excluded from the original tablebase

Test #	Player A	Player B	Player A White	Player B White	Player A - Player B
1	no TB	no TB	124 - 76	127 - 73	197 - 203
2	3 row TB	no TB	157 - 43	65 - 135	292 - 108
3	4 row TB	no TB	173 - 27	38 - 162	335 - 65
4	5 row TB	no TB	182 - 18	35 - 165	347 - 53

Figure 5.3: Wanderer with tablebase playing against original version

5.4 A More Comprehensive Tablebase

In an attempt to both solve 6 $\times$ 6 Breakthrough and see improvement in the tablebase version of Wanderer, a new tablebase with a different limiting factor was created. To accommodate memory limitations, a piece limit of 6 for both Black and White was initially tried. This tablebase did show an improvement when Wanderer was set to only query for forced wins requiring pieces from the top 4 rows for White. However, when Wanderer was set to query for forced wins requiring pieces from the top 5 rows, the time required to query the additional fifth row led to a drop in performance compared to both the new 4 row tablebase and the original 5 row tablebase, as shown in Figure 5.4. It is believed that the additional fifth row doesn't find a significantly greater number of forced wins that would not already be evident as forced wins simply a single move away from a forced win that requires only four rows. Additionally, since the 5 row tablebase entries are not comprehensive, i.e. forced wins involving more than 6 pieces were not included, there may be cases

where Black has an unknown better forced win that the tablebase would fail to detect. A tablebase with a higher piece limit may lead to better player performance, but would require a machine with significantly more memory. Further solutions using secondary storage may also prove to be beneficial, but disk seek time would then become an issue for any game play involving limited turn time.

Test #	Player A	Player B	Player A White	Player B White	Player A - Player B
1	4 row TB	no TB	191 - 9	16 - 184	375 - 25
2	5 row TB	no TB	191 - 9	29 - 171	362 - 38

Figure 5.4: Wanderer performance with adjusted 6 piece tablebase

Chapter 6

Improving 8 × 8 Breakthrough Play

6.1 Introduction

Since positive results for 6 × 6 Breakthrough were found using a tablebase, it is believed that a tablebase for 8 × 8 Breakthrough would have a similar effect. Since the tablebase for 6 × 6 Breakthrough was completed up to 4 rows without any piece limitations and showed the greatest performance increase, it is likely that a 4 row tablebase for 8× 8 Breakthrough would have a similar performance increase. The difficulty with constructing a 4 row tablebase where the total number of White and Black pieces can range to anywhere between 1 and 16 is that the size of such a tablebase would be too large to be memory resident and take too long to compute.

6.2 Tablebase Size and Performance

Without piece limits, the total number of piece configurations that would need to be considered for a 4 row tablebase would be approximately 1.1×10^{11}. By limiting the total number of pieces to 6 or less, the total number of piece configurations drops to 2.3×10^{9}. While this tablebase would not be comprehensive, it can be generated in less than a week and is less than two gigabytes in size. When Wanderer was tested using this tablebase against the original version of Wanderer, the tablebase version did not provide any advantage, as shown in Figure 6.1.

Test #	Player A	Player B	Player A White	Player B White	Player A - Player B
1	no TB	no TB	106 - 94	113 - 87	219 - 181
2	3 row TB	no TB	94 - 106	123 - 77	171 - 229
3	4 row TB	no TB	97 - 103	113 - 87	184 - 216

Figure 6.1: 8 × 8 Wanderer performance with piece limited tablebase

6.3 Future Work Remaining

With greater memory resources, a larger tablebase with a higher piece limit might lead to better player performance for 8×8 Breakthrough. Other possible attempts to create a tablebase that will lead to performance enhancements may be a four row by seven column tablebase in which the remaining column on the board is assumed to be a ”guard” column that can capture any piece that White may try to move to the seventh column. This tablebase could make two queries for both sides of the board, for forced wins involving columns a through g and then forced wins for columns b through h. The additional double query might however slow down query time to the point where it is simply more efficient to complete the simulation to completion and back-propagate the result. Additional work is also needed to determine the feasibility of using a disk resident storage data structure for larger tablebases like those used by some computer chess playing programs.

Bibliography

[1] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence/ Louis Victor Allis*. Ponsen & Looijen, 1994.

[2] Richard Bellman. On the application of dynamic programming to the determination of optimal play in chess and checkers. *Proceedings of the National Academy of Sciences of the United States of America*, 53(2):244–247, 1965.

[3] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[4] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[5] Roger Eckhardt. Stan ulam, john von neumann, and the monte carlo method. *Los Alamos Science*, 15:131–136, 1987.

[6] Kerry Handscomb. 8×8 game design competition: The winning game: Breakthrough... and two other favorites. *Abstract Games Magazine*, 7:8–9, 2001.

[7] Andrew Isaac and Richard Lorentz. Using partial tablebases in breakthrough. awaiting publication, 2016.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. Estonia, 2006. Univ. Tartu.

[10] David Levy and Monty Newborn. *How Computers Play Chess*. Computer Science Press, Inc., New York, NY, USA, 1991.

[11] Richard Lorentz. *Advances in Computer Games: 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers*, chapter Early Playout Termination in MCTS, pages 12–19. Springer International Publishing, Cham, 2015.

[12] Richard Lorentz and Therese Horey. Programming breakthrough. In *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, pages 49–59, 2013.

[13] Richard J. Lorentz. Amazons discover monte-carlo. In *Proceedings of the 6th International Conference on Computers and Games*, CG '08, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artif. Intell.*, 175(5-6):958–987, April 2011.

[15] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[16] Claude E. Shannon. Programming a computer for playing chess. *Philos. Mag. (7)*, 41(314):256–275, March 1950.

[17] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[18] Ken Thompson. Retrograde analysis of certain endgames. *ICCA J.*, 9(3):594–597, 1986.

[19] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 84–99, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Yizao Wang and Sylvain Gelly. Modifications of uct and sequence-like simulations for monte-carlo go. *CIG*, 7:175–182, 2007.