Comparing extended Minimax and Monte Carlo Tree Search for Hawaiian Checkers

Bachelor's Thesis

Mohamed Shokry
402147

08.06.2023

Supervisor: Prof. Dr. Benjamin Blankertz

Prof. Dr. Marc Alexa

Technische Universität Berlin

School of Electrical Engineering and Computer Science

Institute of Software Engineering and Theoretical Computer Science

Neurotechnology

Abstract

In this research the board game Hawaiian checkers and how to play it best is examined. For this purpose the two game tree search algorithms minimax search and Monte Carlo tree search delved into and compared. Among other extensions the novel dynamic board to support both algorithms and a new way of game tree pruning in minimax search are introduced. Experimental data of implementations competing against each other while playing the game is presented and analyzed. Thereby an attempt of contributing to understanding how to play Hawaiian Checkers better and which use cases the tested algorithms are most suitable for is made.

Kurzfassung

In dieser Bachelorarbeit werden das Brettspiel Hawaiian Checkers und Techniken um es zu spielen untersucht. Speziell werden die beiden Spielbaum Suchalgorithmen Minimax und Monte Carlo Tree Search verglichen. Unter anderem werden neue Erweiterungen wie das Dynamic Board zur Unterstützung beider Algorithmen und eine neue Spielbaum Trimmstrategie für Minimax erfunden. Experimentergebnisse von der Gegenüberstellung mehrerer implementierter Computer Spieler werden präsentiert und analysiert. Ziel dieser Arbeit ist es besser zu verstehen, wie Hawaiian Checkers am besten gespielt wird und zu welchen Anwendungsfällen die beiden konkurierenden Algorithmen am besten passen.

Contents

List of Figures

List of Tables

1 Introduction

Hawaiian checkers is an ancient Hawaiian strategy board game originally called Konane. Matches are played by two opposing parties alternating turns in capturing at least one enemy piece until a player is incapable of capturing and thereby loses. It is rendered the Hawaiian version of checkers for the similarity in captures being executed by jumping with an own piece over an adjacent enemy piece of opposite colour onto an empty field. Furthermore Hawaiian checkers is nowadays often played on a checkered 8×8 board though traditionally bigger non-quadratic rectangular boards were used [Siz91]. Apart from these the remaining few rules differ greatly from checkers like consecutive jumps being optional but only possible in the same direction and the heavily restricted piece movement.

In combinatorial game theory properties of Hawaiian checkers were analyzed and especially narrow rectangular boards consisting of up to 4 rows of arbitrary length have been the subject of attempts to solve the game. Only for boards consisting of a single or two rows, a guaranteed winning strategy has been found for every start configuration [Uit21]. This was achieved by taking advantage of the very early decomposition of narrow boards into smaller independent sub-games of reoccurring patterns [CT02, Uit21]. For boards with three or four rows only specific start configuration have been solved [Uit21]. On bigger sized boards the game is more complex and decomposition occurs later [Ern95] requiring different playing techniques.

Especially since the winning streak of computer chess machine Deep Blue against human chess grandmasters and the world champion[CHJH02], it became evident that game tree search is a suitable approach for games that cannot be completely decon-

structed and solved. It is a class of algorithms that operate on modeled game state graphs to gradually choose moves leading to advantageous game states until a match is won.

There are two common paradigms of evaluating the move utility with tree search [BPW+12]. One attempts to exhaustively explore all expected courses of action up to a certain limit in future steps after which successive developments are estimated with heuristics and then projected back onto the initial move [RM93]. The second paradigm bases decisions on the results of simulating during runtime select courses of play until game termination. Here, instead of depending on domain knowledge, decisions are informed by empirically collected data [CBSS08].

Both paradigms are realized through algorithms relying on concepts attributed to John von Neumann, namely the minimax theorem [vN28] and the Monte Carlo method [Eck87]. The exhaustive heuristic paradigm is implemented by depth-limited minimax search, a deterministic recursive algorithm. It is grounded on the observation that in adversarial two party zero-sum games, one player's gain is the other player's loss. Minimax search assumes the enemy to play optimally and forward plans, expecting the worst case, relying on the minimax theorem. It chooses moves that maximize the minimum heuristic potential of winning that remains after the enemy reacts [KM75]. The second approach is realized through adversarial Monte Carlo tree search, a randomized algorithm that iteratively builds a tree structure carrying play-out results of the preceding moves they were enabled by [CBSS08]. Applying the Monte Carlo method, measured win frequencies in a big set of simulation samples are assumed to indicate underlying proportional chances of winning. Thus this tree in theory should organize knowledge about the desirability of moves. Both algorithms can be extended to include aspects of the other approach.

Depth-limited Minimax search is frequently studied, modified and forms the basis for many specialized algorithms [Mar86, dBP96, Sch89, Sch11, SP96]. It was used to solve checkers [SBK+07], forms the core of Stockfish 15.1, the strongest chess engine [MPT22] as of April 2023 [Stoa, Stob, Stoc] and used to be, and remains for many other

games, the preferred algorithm of choice for computer play.

The younger Monte Carlo tree search is best known for its breakthrough in beating world class human players in Go [SSS+17]; a trait which minimax search could not achieve [GKS+12, CMT]. Also, since Monte Carlo tree search works without domain knowledge other than game rules, it reached the rank of state-of-the-art for general game playing [SHS+18, SMBT21] constituting a step closer to general artificial intelligence.

Even before combinatorial game-theoretic analysis, Hawaiian checkers was of interest for the development of artificial intelligence algorithms [Gyl76]. The application of depth-limited minimax search for playing the game has been recorded and examined three times. The first publication hints at a division of matches into 3 phases, gave a game tree branching estimate and highlights crucial domain knowledge to extract heuristics from [Gyl76]. The second research introduced and compared the performance of a wide array of heuristics based on prior recommendations [Tho05]. The most recent research focused on the considerable benefit of deeper tree exploration and alpha-beta pruning [Hen18]. Only in one publication Monte Carlo tree search was documented playing Hawaiian checkers but without further game specific study [SMBT21].

The purpose of this thesis is to contribute to understanding how Hawaiian checkers is played best and how suitable the game tree search algorithms, including enhancements, are in comparison to each other for exploring the game tree and making good move decisions.

First, the game of Hawaiian checkers is assessed through its rules and specific properties. Then the ways minimax search including game strategies and Monte Carlo tree search play get reviewed together with extensions in adjustment to the task of playing Hawaiian checkers. Novel extensions are introduced. Among them are a speculative way of trimming the game tree for combinatorial games that split into independent subgames and the dynamic board combining updating based move generation and state evaluation. In the results chapter experiment data from different algorithm configurations competing in playing the game against each other are visualized and described. Thereupon in the discussion chapter the results are evaluated by mentioning anticipated

aspects, highlighting new gained insights, comparing results to those of other games and looking at how algorithm performance correlates with certain game properties. Lastly, the completed work is reflected and future research suggested in the conclusion chapter.

2 Methods

2.1 Rules of Hawaiian checkers

Hawaiian checkers is played by two adversaries on a rectangular board divided into usually at least 8 rows and 8 columns. A match begins with the board completely filled with black and white pieces in a checkered pattern, so that every piece begins surrounded by enemies of opposite colour or up to two board edges. To open a match, the player assigned with black removes an own black piece either from one of the corners or the four pieces at the center of the board. Then the player white removes a white piece from a field adjacent to the one just emptied by player black. After these two opening actions both players must take turns in jumping with an own piece over enemy pieces. Black begins. Every jump must fulfill two conditions. The enemy piece to capture must be directly adjacent horizontally or vertically to the jumping piece and behind the enemy piece in jump direction there must be an empty field to land in. If the jumping conditions remain fulfilled in the same direction after landing, the landed piece can choose to continue jumping in the same turn. Captured pieces get removed from the board leaving behind an empty field. In most cases a turn ends after just a single jump with a single capture after which the other players turn begins. When a player has no movable piece left while it is their turn to capture they lose and the game ends [Ern95, Siz91].

On the left side, figure 2.1 depicts the first capture that player black can perform after removing the top left corner piece in the first turn and after player white removes the white piece below it during turn 2. The black piece that landed in the top left corner

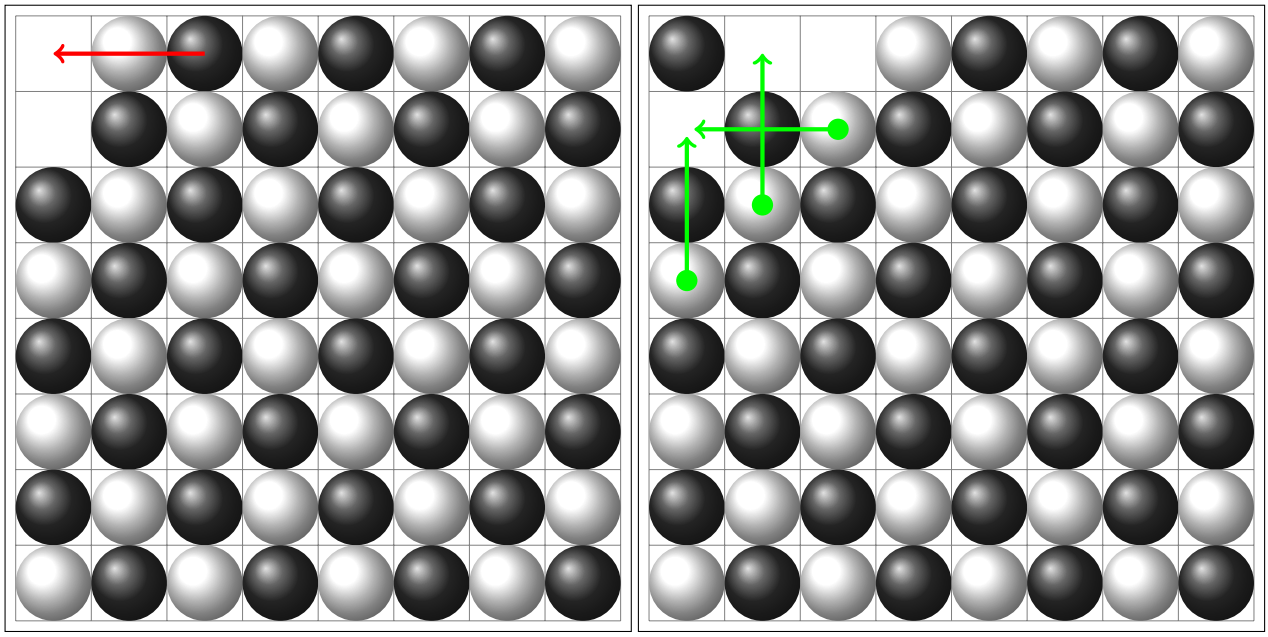

Figure 2.1: The beginning of a Hawaiian checkers early game; turns 3 and 4.

is not capturable until it leaves this field, as next to it, horizontally and vertically, are two edges that prevent jumping and landing in these directions. In turn 4, player white has to choose one of the 3 possible moves indicated by green arrows around the empty area.

2.2 Three game phases and slow game progression

Matches on an 8×8 board usually last between 30 and 55 rounds with the number of fields on the board being a guaranteed limit since in any round at least one piece must be removed from play. Every match progresses through 3 game phases [Gyl76].

2.2.1 Phase one - wake up

Initially, moves can only be done with pieces close to the fields emptied in the two opening turns. All other pieces, though surrounded by sufficient enemy pieces to potentially jump over, remain inactive since in all four directions they are blocked from

landing by own pieces and board edges. But with each piece jumping from the cluttered area over pieces surrounding empty fields, new landing opportunities for both players penetrate the cramped inactive area. Because both the jumping and the captured pieces around the empty area leave behind new empty fields for both players to land in, enabling previously blocked pieces to become active. Thereby in the first phase with each turn both players continue to gain more options to move [Gyl76], preventing the game from ending suddenly. This development can be observed for instance in the change between figure 2.1 during turn 4 and the top left board of figure 2.2 representing the same match after turn 13 where movable pieces for player white are highlighted in green and those for player black in red. Between these two game states, the number of pieces player white can move are more than doubled from 2 to 5 and the number of pieces player black can move are quadrupled from 2 to 8. The number of possible moves for each player to choose from grew even higher. Player white went from 2 to 8 moves to choose from because of one possible triple jump and a separate white piece being able to double-jump. Player black went from 2 to 11 moves to choose from because of 2 possible double jumps and one piece being able to jump in two directions. In this presented case not a single piece capturable in multi-jumps is movable.

2.2.2 Phase two - peak

Throughout the second phase, the increase of movable pieces and possible moves caused by captures and empty fields rippling through the board settles on a peak [Gyl76]. Such a peak can be seen in figure 2.2 in the top right board representing a game state after turn 25. Every section of the board contains empty fields and the number of moves increased to 18 for player black and 11 for white. This value for player black is considerably higher than the average peak branching factor of 10 estimated for 8×8 boards in the earliest [Gyl76] and reiterated in later publications [Tho05, Hen18]. The value for white matches the given estimation. After the peak, what prevents pieces from capturing increasingly is not the lack of fields to land in but instead the lack of adjacent enemy

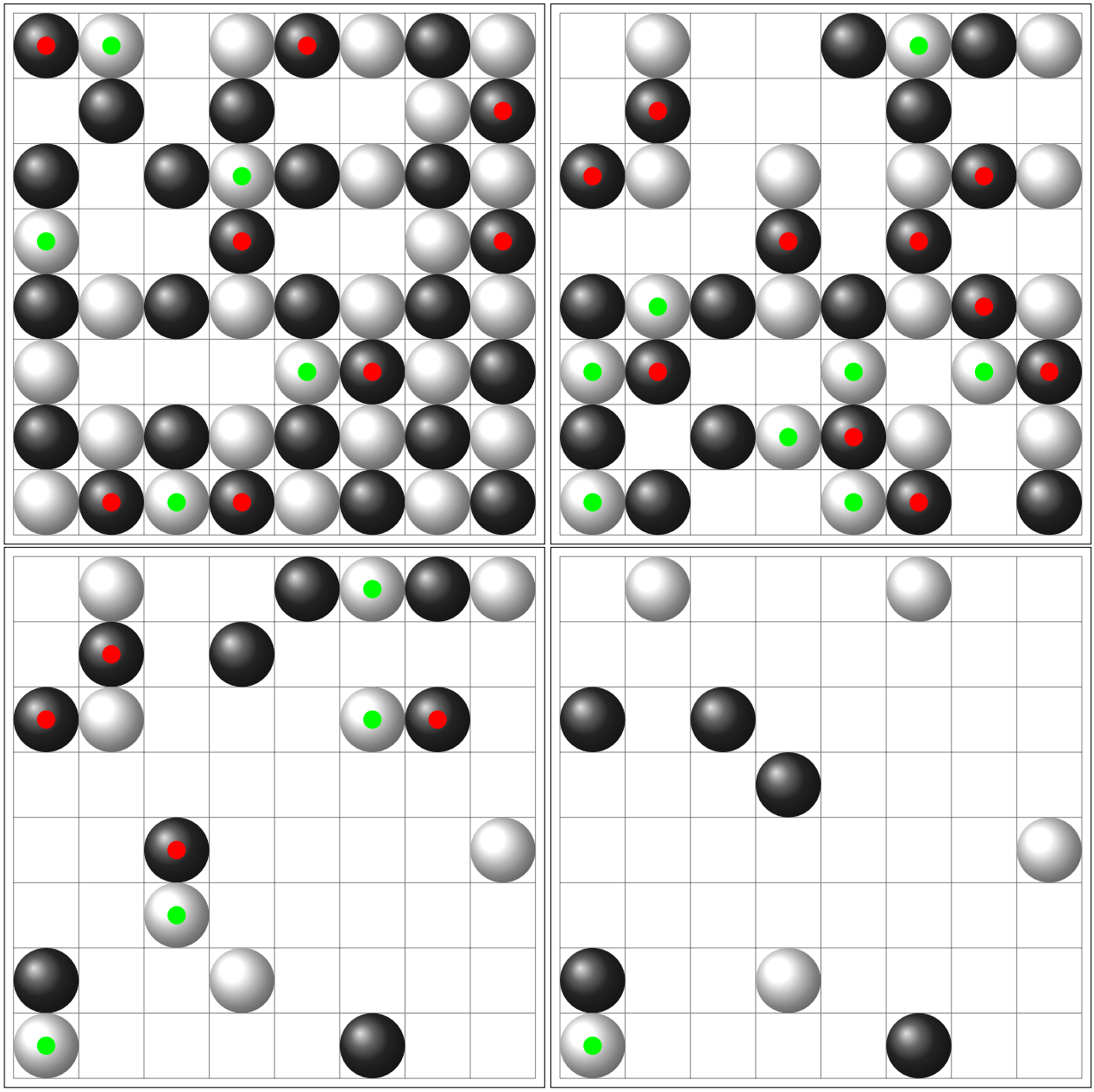

Figure 2.2: Continuation of match in figure 2.1; states after turns 13, 25, 45 and 53.

pieces to jump over since these were captured, moved or were jumped away from in earlier turns. The number of moves to choose from begins to decline and moves on one side of the board rarely affect the other side. The decline in moves only happens gradually due to movable pieces being dispersed on the field and only in rare cases aligned in a straight line capturable through multi-jumps. The maximum number of pieces capturable in a single turn on an 8×8 board is 3 after which consecutive jumps would be blocked by board edges.

2.2.3 Phase three - regression

In phase three after continuous captures the board occupation becomes sparse as can be seen in the board after turn 45 on the bottom left of figure 2.2. Similarly to narrow boards, the game decomposes into nearly and later completely independent subgames that can be analyzed separately[Ern95]. On bigger boards the distance between subgames is bigger, making it even more unlikely for them to intersect. The game can only end when the gradual move decrease leads to one player being unable to perform a single move during their turn. This means that in every subgame there remains no movable piece for them, as is the case for player black after exhausting their last move in the bottom right board in figure 2.2. Up to the point where a player exhausts all their moves no shortcuts can be used to enforce quick wins.

2.3 Optimal play and limitless minimax search

While the game rules are few and simple, winning against intelligent adversaries takes thorough examination of possible outcomes resulting from chosen moves. Even when focusing on immediate gains and ignoring consequences, advantageous moves are not obvious to recognize. There are no pieces of inherent special utility that are desirable to protect and capture because pieces only differ by their colour and position on the board. Every piece capture not only results in the enemy having one less piece to move with

but also in oneself having one less piece close by to jump over to capture in the future to continue playing. Also own executed moves might sabotage other own moves by occupying landing fields other pieces could have needed. Furthermore the naive greedy capture board game strategy of performing the move leading to most captures has not been proven to perform better than playing randomly [Tho05]. Moreover, there are no collectable points for specific achievements that are summed up to a final score. Instead, getting the enemy closer to fulfill the game termination condition of Hawaiian checkers before oneself must be at the heart of every intelligent move. Thus useful greedy strategies could only be implemented by considering factors extracted from domain knowledge relevant to winning. Those strategies must lead the match to a state where the opponent has not a single movable piece, even if he possesses plenty more pieces left on the board than oneself. One approach is to focus on maximizing instant increase or at least minimizing instant decrease of the number of own move options for the next turn. While this might suffice to perform better than a random player, intelligent adversaries can easily counter such short sighted behaviour by planning several steps ahead [Hen18].

Because of the deterministic valid move constraints and game states only changing according to the deterministic move execution, both parties in theory can acquire perfect information about game states and their transitions in Hawaiian checkers [Myc92]. In game theory this information is modeled as a layered node tree [dBP96]. The root represents an initial state, child nodes in lower layers represent following states reachable by executing moves corresponding to incident edges from parent nodes and leaf nodes represent terminal game states. In zero-sum games the gain of one player is the other player's loss, so while the side to move at the root node tries to maximize its gain, in the next layer the other player tries to minimize that same value. According to the minimax theorem of John von Neumann, optimal play against an enemy that makes no mistakes is to perform at any game state the available move which maximizes the minimum remaining possibility of reaching a winning terminal state despite the enemy responding optimally [vN28]. To predict such optimal enemy decision making and resulting re-

maining possibilities of winning, analogous considerations have to be made from the enemies perspective. This leads to a recursive application of the minimax theorem for potentially all possible moves following the initial game state alternating between both players perspectives until the recursion reaches a depth where the choice is between terminal game states implying a loss or win. Choices made at the deepest layers of recursion are filtered and propagated back up if the player that made the recursive call one layer above following the minimax theorem would prefer the outcome over that of any other move. If one of the recursive minimax calls for a move at the root returns that despite optimal enemy response a win can be achieved, the caller knows that by choosing that move and repeating the procedure in the next turn they are guaranteed to win. If no move is guaranteed to win and the enemy plays optimally, the caller is guaranteed to lose the match. Hence, perfect forward planning that works even in the worst case in Hawaiian checkers is equal to solving the game by finding out by means of brute-forcing through the game tree whether there always remains in future a sequence of turns that guarantees the possibility of winning up until the end.

This procedure can be called limitless minimax search and although it works in theory, unlimited minimax search is not practically applicable in the context of computer Hawaiian checkers. The game tree of Hawaiian checkers grows exponentially in width through the possible move choices with every step taken in depth [Ern95, Hen18]. This exponential growth over several dozen turns per match creates a search space that is too big to explore and process in reasonable time [Kor90, Mar86]. In order to be favorable, a computer game player needs to achieve a high win rate while not exceeding limited processing time budgets. The same is prescribed for adversary programs encountered in game competition. Therefore, it makes sense to give up on optimality by not exhaustively exploring complete game trees but only those parts that yield the most crucial information for decision making against flawed but intelligent enemies. Depth-limited minimax search and Monte Carlo tree search are algorithms developed to generally fulfill these requirements.

2.4 Depth-limited minimax search

Depth-limited minimax search is a heuristic algorithm that works similar to limitless minimax search in implementing the minimax theorem by recursively performing a brute force exhaustive tree search and evaluating move utility values depending on the worst case enemy reaction. The difference is that instead of exhaustively exploring the complete game tree and reaching exact terminal state values only a constrained number of layers above the depth limit is exhaustively explored [Mar86]. So, instead of only stopping recursion at real terminal states depth-limited minimax search backtracks when the depth limit is reached. Since non terminal nodes do not hold information about whether a game is guaranteed to be won or not, heuristics are employed to deliver supplementary state evaluations that are filtered and propagated up during backtracking [RM93]. These heuristics estimate how advantageous specific game states are for players concerning the game progression between evaluated state and game end. They are derived from game specific domain knowledge and contain indicators that do not provide absolute certainty but in experience correlate with successful play.

The listing below contains pseudocode of the mini() and max() function that enable the alternating recursion from two perspectives for minimax search. These functions could be used to compute the least disadvantageous worst case outcome reached from a maximizing game state S in n turns by making the following function call: max(S,n). Usually to choose the best move possible from a state, an outer handler function miniMax() similar to the max() function but without termination conditions would be used. Like max() this miniMax() function would repeatedly apply possible moves, call the adversary perspective recursion mini() for the resulting states, remember the highest returned value and additionally the move which lead to that highest value. The realEvaluation() and heuristicEvaluation() functions are supposed to return bigger values to indicate an advantage for the maximizer and smaller negative values to indicate an advantage for the minimizer. The most significant evaluations in size are returned by the realEvaluation() function such that real value terminal node evaluations are weighted

heavier than heuristic estimations and earlier terminal values are weighted heavier because less volatile turns must pass before they can be reached.

```
int max( GameState currentState,  int depthLimit ) {
   if ( isTerminalState(currentState) )
         return realEvaluation(currentState);
   if ( depthLimit == 0 )
         return heuristicEvaluation(currentState);
   int max = -infinity;
   List validMoves = generateMoves(currentState);
   for (Move m : validMoves) {
      currentState = executeMove(m, currentState);
      int score = mini(currentState, depthLimit - 1 );
      currentState = reverseMove(m, currentState);
      if( score > max )
            max = score;
   }
   return max;
}
int mini( GameState currentState,  int depthLimit ) {
   if ( isTerminalState(currentState) )
         return -realEvaluation(currentState);
   if ( depthLimit == 0 )
         return -heuristicEvaluation(currentState);
   int min = infinity;
   List validMoves = generateMoves(currentState);
   for (Move m : validMoves) {
      currentState = executeMove(m, currentState);
      int score = max(currentState, depthLimit - 1 );
      currentState = reverseMove(m, currentState);
      if( score < min )
            min = score;
   }
   return min;
}
```

2.4.1 Strategy and heuristics

Since heuristics replace the real terminal values and are the only source of evaluation for states and thereby moves in the early and middle phase, the accuracy of heuristics can make or break the application of minimax search. In Hawaiian checkers sufficient leveragable indicators that hint at good play exist to let heuristics contribute well to minimax search [Gyl76]. In section 2.2 about the game phases and slow game progression, it was discussed how players can not enforce an abrupt win and must prepare to pass through all mentioned game stages. Getting the enemy to lose means to sabotage their moves and make them exhaust remaining ones while preserving one's own moves and generating new ones if possible. It is strategically of benefit to amass a bigger number of moves than the enemy in the first game phase if after the peak both players lose moves at the same rate, meaning that the enemy exhausts their smaller number of moves earlier and thereby loses. But having more move options during the peak in phase three does not guarantee winning as the example match looked at in figure 2.2 proves. After turn 25, player white had gathered 11 possible moves while player black had aggregated 18 possible moves which are about 1.64 as many. Despite that big disparity after turn 45, both players equally only had 4 movable pieces left and in turn 55 white wins when both sides had exhausted all their moves and it was black's turn to capture. Preserving moves after the peak seems to be of highest priority.

In a survey of 18 different heuristics in 3 groups of 6 heuristics measurements for their minimax win rates for depth limits 1 to 4 against each other and against random players were conducted [Tho05]. The first group consisted of heuristics that only take into account numbers of pieces on the field, regardless of whether these were movable or not. These performed the worst with all except one having a win rate lower than 50% against random players for all search depths. Such results are assumed to occur due to the number of pieces being guaranteed to decrease by at least 1 during every turn, except in the final turn. Additional changes of the number of pieces for either player occur only through multi-jumps thus these heuristics focus on performing many multi-jumps which

has no obvious benefits for the jumping player [Tho05]. Multi-jumps are only better than single jumps when they thwart multiple future opponent jumps and cause harm if they sabotage own possible jumps. Both effects are paid attention to when focusing heuristics on counts of movable pieces or moves. The second group focused on the number of possible moves for either player and the differences between them calculated through either subtraction or ratios both weighted and unweighted. These heuristics performed much better than group one with most having a win rate between 75% and 90% against random players and two having reached an average win rate of over 50% against other heuristic players. The best performing heuristics group is that focused on the numbers of movable pieces for both parties. These heuristics in most cases perform better than their counterparts in group two. Also their win rate measurements are the most consistent over different search depths. The most successful evaluation function with a win rate of 61.96% against heuristic players and up to 91,2% against random players was heuristic q.

q = number of own movable pieces ÷ number of opponent's movable pieces.

We use evaluation function q because of its highest win rates and because it pays attention to the own number of moves, the enemy number of moves and the ratio between them. Through this heuristic the minimax algorithm is guided to not only focus on gaining many moves or focus on sabotaging enemy moves, but on increasing the gap between the own number of movables and the enemy number of movables as well. This is also of benefit, considering that movable pieces are easier to preserve than many moves since focusing on having many moves means focusing on having pieces that can move in multiple directions or perform multi-jumps. Once such a piece is moved, several moves are lost at once which could send misleading signals through the heuristic.

2.4.2 Alpha-beta pruning

While the win rate is mainly dependent on the quality of heuristics and the search depth, the minimax algorithm can be enhanced from multiple angles to lower its runtime and

enable deeper more informative searches. The most significant enhancement minimax search is almost always implemented in combination with is the alpha-beta pruning optimization [Sch11]. Alpha-beta pruning speeds tree searches up by cutting off subtrees that are not required for minimax evaluation. By trimming irrelevant subtrees less nodes have to be explored and therefore the algorithm search time can be reduced without influencing the final decision quality [FGG+73]. This is achieved by reasoning like in branch and bound but for adversarial two party trees where the perspective from which the decision to cut is made from alternates with every tree layer between the opposed game parties. For both perspectives separate bounds are kept.

The bound applied from the perspective of maximizing nodes is called alpha and represents a lower bound which is increased by maximizing nodes every time children propagate up values bigger than alpha. If no alpha bounds were inherited by a maximizing node from higher nodes, he, by default, sets the value to -infinity, so that the first value propagated up from lower mini()-calls becomes the first measured alpha lower bound. The alpha bound is passed down during recursion and every time a minimizing node, that inherited an alpha bound from layers above, gets returned from a max()-call a value smaller than the alpha lower bound all other child nodes of the minimizing node can be cut off from exploration. The alpha-cutoff can be safely done because maximizing nodes in upper layers that sent down the alpha bound have no reason to perform moves that lead to the minimizing node which can enforce game state values lower than the inherited alpha. This is because the maximizing player can instead choose to use the move performed before the mini()-call that lead to a higher worst case (for the maximizer) state value which lead to setting the alpha bound value higher.

Analogously, the bound applied from the perspective of minimizing nodes is called beta and represents an upper bound which is decreased by minimizing nodes every time children propagate up values smaller than beta. If no beta bounds were inherited by a minimizing node from higher nodes, he, by default, sets the value to infinity, so that the first value propagated up from lower max()-calls becomes the first measured beta upper bound. The beta bound is passed down during recursion and every time a maximizing

node, that inherited a beta bound from layers above, gets returned from a mini()-call a value bigger than the beta upper bound all other child nodes of the maximizing node can be cut off from exploration. The beta-cutoff can be safely done because minimizing nodes in upper layers that sent down the beta bound have no reason to perform moves that lead to the maximizing node which can enforce game state values higher than the inherited beta. This is because the minimizing player can instead choose to use the move performed before the max()-call that lead to a lower worst case (for the minimizer) state value which lead to setting the beta bound value lower.

2.4.3 Move ordering

When the alpha bound is raised and the beta bound lowered more redundant subtrees are cut off from tree search [Sch89]. Early tight bounds lead to cut offs happening in the higher layers of the tree and result in many nodes being cut off at once. By exploring the strongest moves for players first an order is achieved that sets the tightest bounds in the earliest subtree explorations leading to substantial time savings being expected [Mar86, Sch11].

Like recommended in earlier studies as future work [Hen18], move ordering to support alpha-beta pruning in Hawaiian checkers is designed in this work for the first time. This move ordering is employed for every minimax recursion; therefore, the most significant moves can not be identified by adding even more deep minimax searches for every move. Instead, inspired by the used heuristic, the instant effect of every move is evaluated by predicting how the game state changes for both sides. Effects, weighted with a point per occurrence, are divided into two groups. The first group is weighted with positive points:

1. The jumping piece was neighboring an opponent piece in other than the jumping direction behind which was an own piece, therefore the own piece was enabled to jump in a new direction.

2. The piece enabled to jump in a new direction in the preceding effect was turned

from an unmovable piece into a movable piece.

3. The neighboring opponent piece was able to capture the piece that just jumped, therefore an enemy jump was thwarted.

4. The thwarted move in the preceding effect was the only move possible for that enemy piece, therefore a movable enemy piece was turned unmovable.

5. A piece captured while jumping was capable of capturing own pieces, therefore an enemy jump was thwarted.

6. After landing the jumped piece is adjacent to a capturable enemy piece in other than the jumping direction, therefore the performed jump enabled a new one for the same piece.

The other instant effects are weighted with negative points:

1. The jumping piece after landing is capturable by enemy pieces.

2. The enemy pieces in the preceding effect were turned from unmovable pieces into movable ones.

3. The jumping piece after landing blocks preceding jump possibilities for other own pieces.

4. The own pieces blocked from jumping in the preceding effect had no other move options and therefore were turned unmovable.

5. Pieces captured while jumping enabled move opportunities for own pieces, capturing them thwarted own moves.

6. The own pieces that lost the thwarted moves in the preceding effect had no other opportunities and became unmovable.

For each move to be ordered the points for all of its effects are are summed up to an order score. The final move list which is provided to minimax search with alpha-beta

pruning will be ordered by descending order scores. This is similar to ordering based on a shallow tree search of the first layer but computationally cheaper as it capitalizes on jumping conditions heavily confining the movement of pieces. Instead of executing a move, scanning the whole board for state evaluation and then reversing the move, only those parts of the board that are affected by the move are selectively analyzed in a predictive manner.

2.4.4 Dynamic Board for move generation and state evaluation

The valid move generation including move ordering for every minimax recursion, as well as the heuristics used for evaluation of depth limit nodes, constantly depend on information about movable pieces. While completely scanning the game board once takes only minute computation, repeating it anew many ten thousand or even hundred thousand times significantly slows down minimax search [Bra82] redundantly, since the results only differ slightly in between succeeding board scans, while most of the computed data remains the same. A dynamic data structure that cumulatively prepares data about movable pieces can be maintained through minor updates after every executed move.

The naive board representation that only records whether fields are occupied or not is enhanced to the novel dynamic board for incremental move generation and state evaluation. For each field, this new game board, in addition to its occupation, if occupied also records all of the possible moves for the occupying piece. It also implicitly stores a list of all fields that hold movable pieces for either player by fields in the list holding links to preceding and succeeding fields carrying movable pieces. Every time a move is executed, for each field emptied or landed in during the move, theoretically affected occupied pieces in all four directions are informed so that the data they hold is updated. In addition to the possible moves memorized in fields and the two implied linked lists, statistics that count movable pieces and their moves are simultaneously updated when the lists are changed. In this way, crucial data needed during every minimax recursion

and evaluation function call can be accessed with only a small computation cost that occurs in between state transitions.

2.4.5 Window Subgame Pruning

Among the methods for minimax tree trimming are those that are guaranteed to cut off only redundant subtrees as done by alpha-beta [KM75] and there are plenty of ways of performing speculative cut offs that might affect search outcomes. Since depth-limited minimax search does not operate on exact real values but approximations provided by heuristics taking the risk of speculative trimming can be rewarding enough [Bjo02] and the saved search time can be used at other places for deeper search that might improve search quality [Hen18]. This is because by conducting the tree search deeper, the distance between the search horizon evaluation and real terminal nodes becomes smaller. Thus, the degree of speculation implied by using heuristics shrinks, as the expected causality between perceived advantages that are normally correlated with winning and achieving a win would be strengthened by the possible game volatility decreasing by less turns remaining before the game ends.

Like other combinatorial games, Hawaiian checkers breaks into subgames that can be played independently [Ern95]. Winning a match means always having a subgame left to move in, hence, to play a subgame well is to focus on increasing the number of own move options inside the subgame while making the opponent exhaust theirs, as if it were a normal complete game. Moves that happen on the board outside of the subgame are expected to not change the number of movable pieces inside the subgame; therefore, whichever moves are executed outside of it and in which order is expected to be irrelevant for it. As a consequence of this independence, the total minimax game tree can be divided into separate subgame trees. The sum of these subgame trees is much smaller than the combined total game tree because these subgame trees maximally have as many layers as pieces inside the subgame accounting for divided exponents and branch only with the number of possible moves inside the subgame accounting for

divided bases of divided exponential tree growth in width with depth. That means that during an own turn, when a move has to be executed, instead of on the combined total game tree, minimax search is separately conducted on each subgame tree either in a shorter combined time or each with a deeper search limit. Then, following the preferred heuristic, that move is chosen which in its subgame leads to the biggest increase of the ratio between the own and enemy's number of movable pieces.

To mimic this divide and conquer type approach to playing Hawaiian checkers even before the game decomposes, the novel Window Subgame Pruning technique is invented in this work. As discussed in the game phases section, moves in Hawaiian checkers that happen on one side of the board hardly affect pieces on the other side of the board except in the early game where the chain reaction of empty fields spreading begins. So when a player has to choose a move during their turn, for each move Window Subgame Pruning draws a window of preset inner padding around the straight line the moved piece would traverse during move execution. Recursive minimax calls below each of these moves are limited to their corresponding window, treating it as if it were the complete game board. This change to the way minimax search operates can simply be implemented by ignoring all moves provided by valid move generation that would be possible for pieces outside of the window and thereby pruning these moves. The evaluation function remains the same with the difference that it is not only called when the depth limit or real terminal nodes are reached but also when the side to move has no movable piece left inside of the window. The tighter windows are set the more moves and subsequently search subtrees are pruned. But tight windows increase the relative influence that pieces which unexpectedly jump into the window from the outside have, just like a single piece on a complete 8×8 board on average has a bigger influence on state evaluation than single pieces on a 20×20 board. This bigger relative influence of unexpected changes would lead to volatile evaluation which could negatively influence the consistency and quality of data collected during minimax search. Just like with any other speculative pruning approach a trade off between search time reduction and search quality has to be regulated. Bigger windows are drawn around multi jumps. This neither

implies them being preferred nor avoided but instead widens the range of values that could theoretically be assigned to them by heuristics since more opponent's but also more own movable pieces could theoretically fit into bigger windows.

2.4.6 Obstacles and limits

In summary the only obstacle that prevents the exhaustive limitless minimax search from being applied from the beginning on and depth-limited minimax search from deepening exploration is the Hawaiian checkers game tree exponentially growing with every layer in width. Even the most sophisticated pruning techniques that avoid high risks of compromising on search quality do not suffice to reach terminal nodes through minimax search in the early and middle game. The accuracy of the worst case assessment and resulting move choice according to the minimax theorem is proportional to the accuracy of heuristics in judging whether game states are advantageous for players. In conclusion minimax search is limited by the currently not in feasible time conquerable tree size and by the game insight of those who implement heuristics.

2.5 Monte Carlo tree search

Since complete certainty is given up in Hawaiian checkers probabilistic techniques to play the game are worth being examined. Monte Carlo tree search is a stochastic tree search algorithm that iteratively constructs a preserved tree of game simulation result samples used to inform game state and move evaluation [BPW+12]. In isolation individual random simulations until terminal states hold little informational value about move utility and the likeliness of opponents using certain moves. By repeating them, a substantial amount of time following the Monte Carlo method patterns of correlation indicating causality between the execution of certain moves and winning are expected to become apparent. In essence, the algorithm estimates the winning probability of moves based on the frequency of wins occurring in simulations of the game progression fol-

lowing these moves. Which moves to test and gather simulation data about is regulated by a constructed tree policy that regulates both the exploitation of the most promising moves and sufficient exploration of their surroundings to avoid overly speculative extreme selectivity [CBSS08]. In the beginning, measured win rates fluctuate more but over time they stabilize because new single simulations have little weight in comparison to the bulk of preceding ones. The longer the algorithm runs, the longer it takes for move prioritization to change.

2.5.1 Four phases

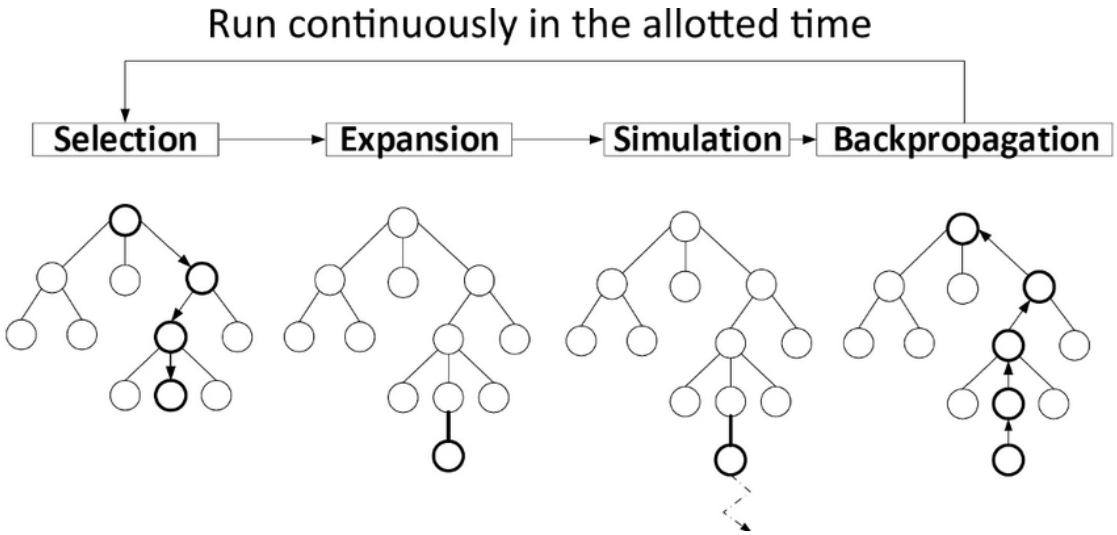

Figure 2.3: One iteration of the Monte Carlo tree search algorithm [ŚGSM23].

When Monte Carlo tree search is first applied, the constructed sample data tree starts with a single root node that represents the Hawaiian checkers game state of which the best move is to choose from. All of these moves are considered unexplored since the states they lead to are not yet added to the constructed tree. In addition to move and adjacent node references, every node stores a visit counter and a points counter, both initialized with value 0 during node instantiation. Points are given out of the perspective of the root node. Benefits for the opponent are implied by increased visit counts without an increase of the node points count. Therefore, the points to visit counter ratio is the mea-

sured win frequency out of the root player perspective and 1 minus this ratio is the win frequency out of the opponent perspective. After tree initialization the algorithm works by continuously performing consecutive iterations composed of the 4 steps outlined in figure 2.3: selection, expansion, simulation and backtracking [CBSS08]. Iterations are discontinued when a prescribed number of repetitions or a time limit is reached. Then out of all moves possible from the root node the one that leads to the biggest points by visits ratio is chosen as the best move to perform.

Selection - UCT policy

In the selection phase starting from the root node the tree is traversed into deeper layers until a node with unexplored possible moves is reached. In the common Monte Carlo tree search variant for each tree layer, the lower node navigated to next is determined through the Upper Confidence bounds for Trees, in short UCT, policy. It treats the choice between child nodes attributed with measured win rates like the choice between bandit machines in the multi-arm bandit problem [KS06, KSW06]. The multi-arm bandit problem of balancing the exploitation of the highest measured reward frequency bandit and the exploration of other bandits to improve the quality of Monte Carlo simulation based reward probability approximation is commonly solved by choosing the bandit with the highest UCB1 value. The UCB1 formula for a bandit j whose arm was chosen to be pulled in n_j times out of n total bandit pulls which lead to a measured reward frequency r_j [BPW+12]:

$$UCT_j = r_j + \sqrt{\frac{2ln(n)}{n_j}}$$

The first component of the formula grows through higher measured rewards per pull and promotes exploitation. The second component shrinks by the number of pulls in comparison to other bandits and thereby promotes exploring less tried bandits. When applied to game trees, the UCT formula further regulates the trade off between exploitation and exploration by multiplying the second component with the constant $2 \cdot C_p$. For applications of UCT in Monte Carlo tree search with terminal evaluations ranging be-

tween 0 and 1 it is recommended to set C_p to the value $\frac{1}{\sqrt{2}}$ [BPW+12]. After applying this recommended constant value, the formula is simplified and turned into UCT functions UCT_{root} and UCT_{opp} where the win frequency component is adjusted to the two player perspectives. The functions for evaluation of successor nodes s that out of n total selections done by their parent were selected and visited v_s times leading to p_s points:

$$UCT_{root}(s) = \frac{p_s}{v_s} + \sqrt{\frac{4ln(v)}{v_s}}$$

$$UCT_{opp}(s) = \frac{-p_s}{v_s} + \sqrt{\frac{4ln(v)}{v_s}}$$

In layers for the root the move and resulting opponent state is selected for which UCT_{root} is maximized. In adversary layers the child node which maximizes UCT_{opp} is selected and descended to. When a node with unexplored moves is selected the expansion phase follows. The runtime complexity of the selection phase scales with the constructed tree depth multiplied with the game tree branching factor of moves picked by the UCT policy each layer.

Expansion

In the expansion phase a random unexplored move of the selected node is explored by creating for it a new child node that represents the resulting state. So with each iteration the tree is expanded by one node after which the simulation phase begins. Since a new set of unexplored moves has to be generated for the expanded node the runtime complexity of this phase is that of the move generation.

Simulation

The simulation phase is where the samples of complete play-outs for the grand Monte Carlo simulation are taken. In a game copy, starting from the last expanded node, random moves are performed to simulate the remaining game up to a terminal state. If the last turn, where a player was unable to move, belonged to the root node, the simula-

tion returns the value 0 for loss and if the side to move in the terminal state is root's opponent, then the simulation returns 1 to signal a win. This phase has the runtime complexity of random move generation multiplied by the number of remaining turns between the expanded state and terminal states.

Backpropagation

In the backpropagation phase the simulation value is added to the points counters of the expanded node and all nodes in the selection phase path. Additionally, the the visit counters of all of these nodes are each incremented once. Then the next iteration can begin. The runtime complexity of the backpropagation grows linearly with the selection path length.

2.5.2 Subtree inheritance

Since lower nodes in the sample tree constructed by Monte Carlo tree search correspond to future game states [CBSS08], subtrees in lower layers can be treated like individual constructed trees for future states. Instead of constructing during every turn a new tree, subtrees can instead be passed on from the construction that happened in the last own turn. As the UCT policy attempts to prioritize exploring the moves that are most likely to be used, the corresponding subtrees following them are the best developed, whereby the headstart through subtree inheritance is increased and especially so if after the own executed move the opponent chooses the move that was predicted for him by UCT. So the better Monte Carlo tree search performs the higher the additional headstart.

The inherited tree having been constructed as a subtree through the same policies in earlier rounds does not change the informational value of the held sampling results. Because individual simulations from the same node are rolled out independently from how the start node was reached and instead differ only depending on the moves performed during the simulation itself.

2.5.3 Dynamic Board

Like the minimax search the Monte Carlo tree search is constantly in need of generated valid move possibilities depending on game states. The same dynamic board extension used to enhance move generation for minimax recursions can be employed to speed up the simulation phase where a valid move has to be generated the number of turns it takes to reach terminal states and also to speed up in the expansion phase the unexplored moves generation for the new node. To support the simulation phase it is not necessary to generate all move possibilities; a single random move picked out of the incrementally updated move list in the Dynamic Board suffices.

2.5.4 Flexibility

The presented Monte Carlo tree search together with UCT can be implemented without any prior playing experience or more than superficial game specific knowledge [RSS12]. That makes the technique flexible and easy to use for games consisting of overwhelmingly many interknitted problems that are hard to unravel. Conditions to be fulfilled by use cases for the applicability of Monte Carlo tree search are the following three [Cha10]. Firstly, the little game specific knowledge needed must contain game rules, so that future game states can be modeled in the constructed tree and simulations that recognize terminal values can be conducted. Secondly, since simulations are played out up to terminal states, it must be guaranteed that all lines of play are either finite or recognizable as endless and neglectable. Otherwise, a single endless simulation could block the algorithm from working. Thirdly, terminal state evaluations must be reducible to a bounded range of values so that UCT is not completely dominated by unexpectedly high values that throw the policy off. Hawaiian checkers, like most board games, fulfills all of these conditions.

2.5.5 Limits and efficiency

In theory Monte Carlo tree search with UCT, if allowed to run long enough, is proven to construct a game tree that is equivalent in move evaluation to optimal limitless minimax [BPW+12], although such lengthy computations equally are not practically applicable. For a Monte Carlo tree search to be effective, the most important factor necessary to occur within computation budgets is a sufficiently high number of single simulation samples under the moves to evaluate together with reasonably discoverable long-term impacts of these single moves [Win17, Coo21].

If the number of taken samples is too small in relation to the likely to be reached parts of the search space, the aggregated Monte Carlo simulation lacks the data to convolute an accurate approximation of the game progression into win rates for moves in the root layer. The exception is if specific frequent consequences are triggered indicating strong long-term effects of moves and therefore predictablity. If it were known that this is the case, it could be more thoughtful to use an heuristic algorithm that predicts the same conclusion which simulations would have to arrive at after starting every time from scratch.

If moves had no long term impact at all, neither on the game progression nor on a final score, tree search would not need to be employed in the first place. So, when tree search is conducted it is always assumed that all moves to evaluate have a long term effect varying by intensity.

If long term effects of moves do exist but are too small, it might take too long for the Monte Carlo simulation to overcome statistical noise in measured move win frequencies that emerges because of the variance in enabled playouts and final outcomes. If it were known that this is the case, since adversaries would have the same problem, it could be more thoughtful to at least use an algorithm that exhaustively explores the highest search tree layers for short-term effects of moves. Moves are guaranteed to at least have short-term effects because otherwise all matches would progress and terminate identically.

3 Results

Here, experimental data of computer Hawaiian checkers players competing against each other is presented in three sections. Player programs were implemented as java classes based on discussed techniques [Jav]. The first results section only includes extended minimax players. Then, two Monte Carlo tree search based players are compared. In the last results section, minimax is set directly against Monte Carlo tree search. The experiments were conducted by making programs alternate in choosing moves to execute until game endings are reached. Elapsed time per turn and winners of completed matches were recorded to enable putting into proportion how much time it takes for the players to identify relatively decent moves. To assess the effects of extensions further measurements were taken depending on their base algorithm and intended improvements. Match-ups and result visualizations are arranged in a way that facilitates reviewing how the extensions affect elapsed computation time and win rates.

3.1 Competition of minimax players

In this experiment group the three minimax players without speculative pruning `AB`, `DAB` and `DABS`, stand at the same time against a window subgame pruning player `WDABS`. The names are acronyms referring to included extensions. The benchmark player `AB` and all others include alpha-beta pruning `AB`. The dynamic board is indicated by a `D`. `S` stands for move ordering and `W` for window pruning.

Since `AB`, as well as `DAB` and `DABS` always make the same decision and only vary in how they reach it, differences in their performances are measured by calling them

each, one after another, in the same turn for the same game state while measuring how long it takes for them to arrive at the same outcome. Then the move chosen by all three is executed only once per turn against `WDABS`. As the algorithms on both sides are deterministic, matches started with different early game states and with switching colour assignments to prevent all matches from turning out identically.

3.1.1 Experiment 1

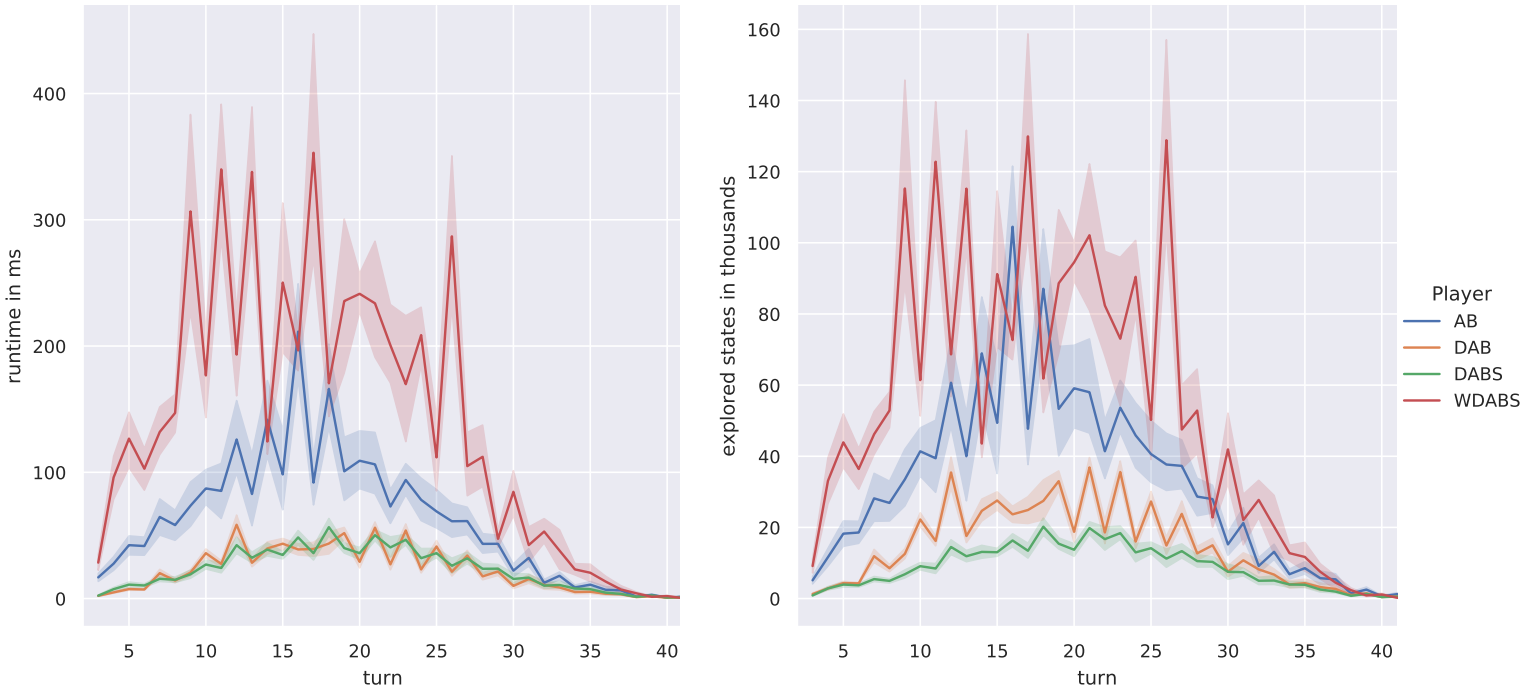

Figure 3.1: On an 8×8 board AB, DAB and DABS search to depth 6; WDABS searches to layer 8 with 4 fields window padding.

Figure 3.1 shows results of the non-speculative players searching the game tree of an 8×8 board, 6 layers deep against `WDABS` with a search depth limit of 8 layers and an inner window padding of 4 fields in each direction. The chart on the left side shows the elapsed time players needed per turn in milliseconds while the chart on the right side provides the number of explored search tree nodes in thousands and thereby recursions

called by players per turn.

WDABS explored the most nodes, taking almost consistently the longest computation time per turn; between turns 15 and 20 up to three and half times more than AB. Out of the 100 conducted matches WDABS won 53 and the other three players won slightly less with 47 wins. Both DAB and DABS were faster than AB despite searching equally as deeply. DABS explored less nodes than DAB but it did not run noticeably faster.

Figure 3.2 shows at which index the moves chosen for execution by players was provided by the move generators they depend on. The highest index at which a best move was found is 16. To avoid the diagram from being cluttered, only indices 0 to 4 are included. The move ordering of DABS was the most reliable in providing the move that would be evaluated the highest at the first position in generated move lists.

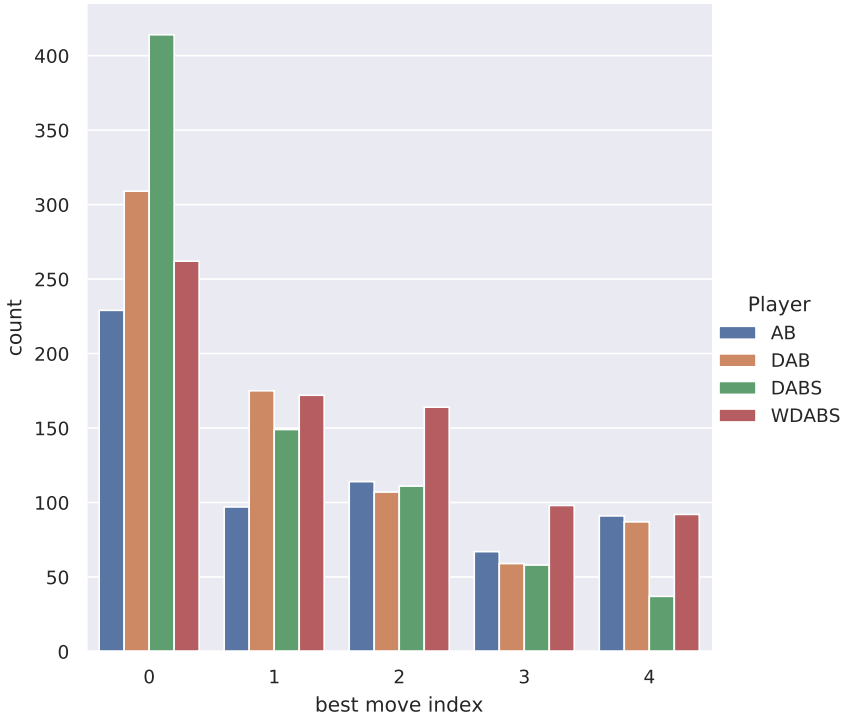

Figure 3.2: Where did move generation place the best move between turn 10 and 30?

3.1.2 Experiment 2

Experiment 2, of which results are shown in figure 3.3, was set up like experiment 1 on an 8×8 board. Here, however, the depth limit of AB, DAB and DABS has been increased to 7 layers, while the inner window padding of WDABS has been widened to 5 fields in each direction. Again, the chart on the left side displays elapsed time per turn in milliseconds and the chart on the right side shows the number of explored states in thousands.

Figure 3.3: On an 8×8 board AB, DAB and DABS search to depth 7; WDABS searches to layer 8 with 5 fields window padding.

AB took the longest computation time per turn and explored the most states, even more than WDABS which searched one layer deeper and compared to the previous experiment with smaller windows explored more nodes. DAB explores similar amounts of states as WDABS but its computation time aligns more with the faster ABS that explores the least states. Out of the 100 conducted matches WDABS this time won only 25 while the slower AB and the faster dynamic board players without window pruning won 75.

Figure 3.4 shows again that the move generation for DABS is superior, closely followed in this case by WDABS and DAB even though DAB is not designed to prepare moves in a specific order before exploration.

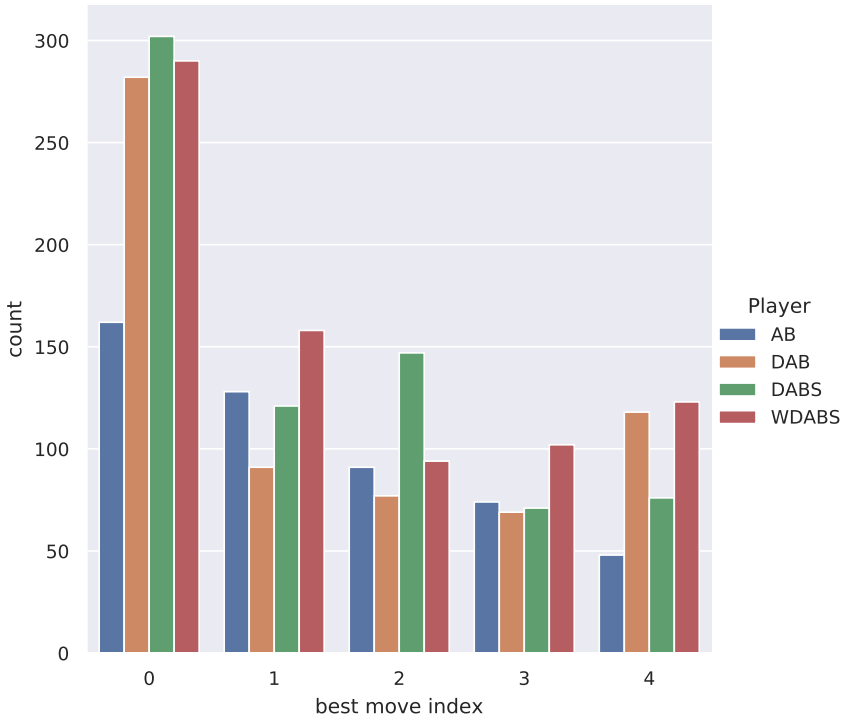

Figure 3.4: Where did move generation place the best move between turn 10 and 30?

3.1.3 Experiment 3

For Experiment 3, board dimensions were stretched to the more traditional [Ern95] 14×17 shape. The target of this change is to look at how player performances scale with growing search trees. For this purpose, the same depth limits and window size as in experiment 1 were adopted. Figure 3.5 now shows runtimes in complete seconds and counts of explored states in millions.

Players can be ranked similarly concerning runtimes and explored state counts as in experiment 2 but with two differences. Firstly, DAB is no longer equally as fast as DABS but rather even faster despite exploring more states. Secondly, WDABS here

Figure 3.5: On a 14×17 board AB, DAB and DABS search to depth 6; WDABS searches to layer 8 with 4 fields window padding.

explored more states than DAB during the early game. The big gap between the runtime of AB and the runtimes of other players is not found in the numbers of explored states. The player with the best scaling runtime is DAB. With 30 wins, WDABS won a few more games in this setting than in experiment 2, though less than in experiment 1 where depth limits were identical. The other three players won 70 out of 100 matches.

Figure 3.6 shows that on this bigger game board the move generation for DABS remains the best in terms of accuracy, leaving that of other players, specifically AB, far behind.

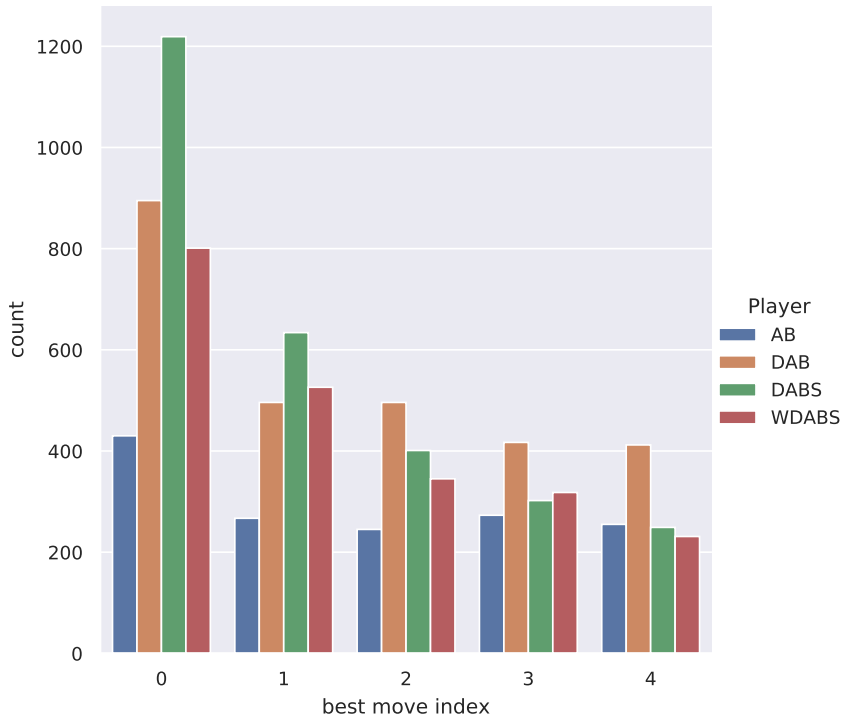

Figure 3.6: Where did move generation place the best move between turn 10 and 120?

3.2 Monte Carlo tree search against Monte Carlo tree search

In this experiment a player `MCTS`, based on Monte Carlo tree search with UCT, plays 100 matches on an 8×8 board against another Monte Carlo tree search with UCT player `DMCTS` that uses the dynamic board extension for random move generation in the simulation phase. Both programs receive a computation budget of 5 seconds per turn and inherit subtrees from constructed trees in previous turns. Figure 3.7, on the left side, shows the number of iterations players managed to perform during their respective turns and the chart on the right side visualizes the through Monte Carlo simulation estimated win rate of moves chosen to execute. Each string corresponds to the measurements taken in a single match for the player with the assigned colour. While using the dynamic board significantly sped up minimax search in all documented cases, `MCTS` manages to

perform much more iterations per turn than DMCTS. For both players the iteration counts grow superlinearly with each turn until either a direct winning move is found or terminal state realized after which iterations are ceased. For both players the estimated best move win ratios during most parts of a match remain between 40% and 60%, though MCTS not only performs more iterations but also exhibits a slightly bigger variation in move evaluations during the early and middle game. Only after turn 25, where the first iteration counts plummet, do move evaluations become more distinct and quickly reach either 0% or 100%. Half of the 100 conducted games were won by MCTS and the other half by DMCTS.

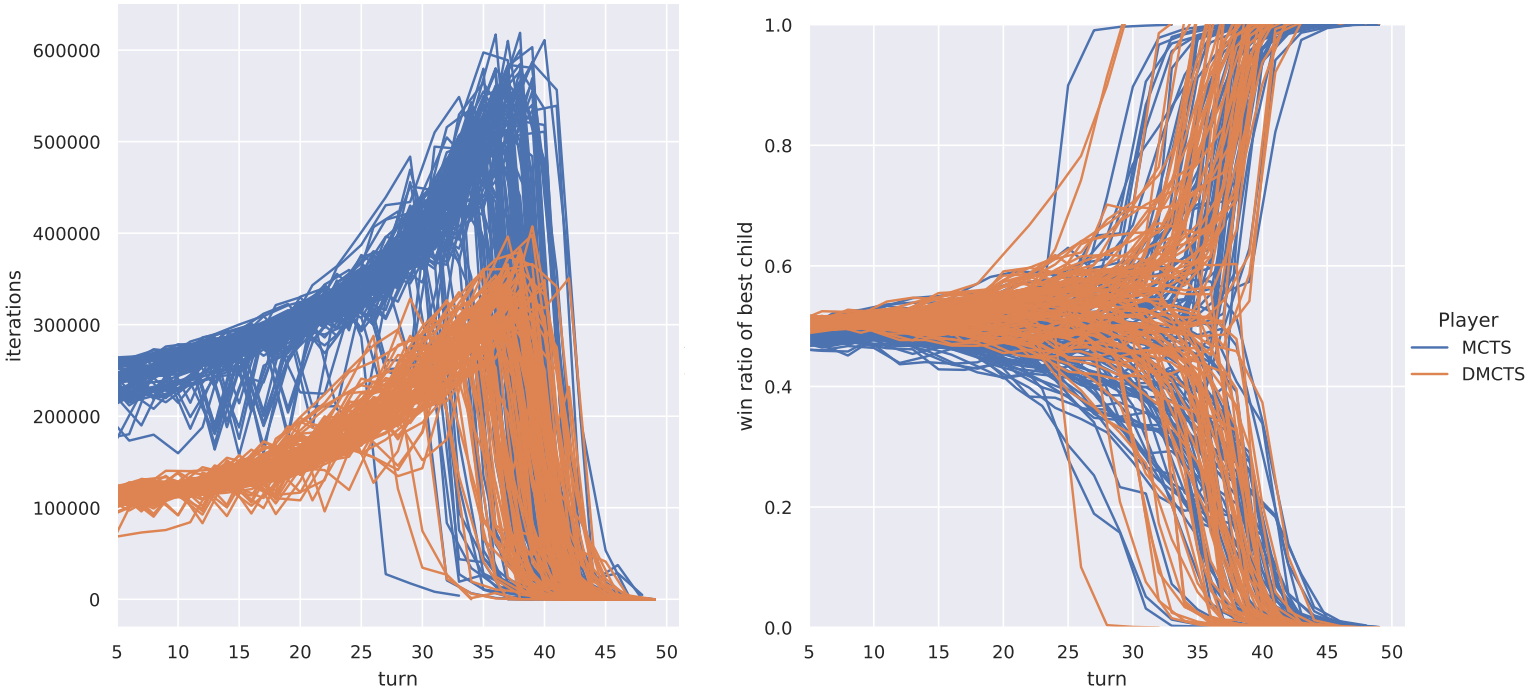

Figure 3.7: On an 8×8 board MCTS and DMCTS play turns of 5 seconds.

3.3 Minimax against Monte Carlo tree search

In this experiment the minimax player DABS from section 3.1 plays against the Monte Carlo tree search player DMCTS from section 3.2. The former completes its search up to

depth limits DL and the time it takes, multiplied with budget multipliers BM, is granted to DMCTS as runtime budgets. Every conducted match-up formed by a distinct combination of DL and BM in fields of table 3.1 was repeated in sets of 100 simulations. Result values are counts of wins achieved from the minimax search perspective. The rightmost column shows the mean of the highest turn runtimes minimax reached for different DL. Budget multipliers are doubled with every step to account for exponential growth of time budgets for DMCTS.

By increasing depth limits in steps of two, minimax appears to run roughly 11 (transition from DL 4 to 6) to 27 (transition from DL 8 to 10) times longer. In each row with increasing BM the number of wins for the minimax search decreases whereby the win count for Monte Carlo tree search increases. The minimax player clearly dominates DMCTS, regardless of generously high budget multipliers. Even a BM of 16 does not suffice for DMCTS to reach the win frequency WDABS achieved against DABS in less time. The highest DMCTS win rates were achieved in match-ups with the relatively shallow DL of 2. Simulations for the crossed out fields in the lower right corner of the table were aborted because of excessively high runtimes.

DL \ BM	1	2	4	8	16	32	64	avg. max t
2	87	85	88	74	76	67	61	2.765ms
4	99	93	93	87	82	74	×	32.160ms
6	97	92	98	92	87	×	×	361.229ms
8	97	91	91	91	×	×	×	4.749s
10	93	92	90	×	×	×	×	126.593s

Table 3.1: Results of depth-limited minimax against Monte Carlo tree search with UCT.

4 Discussion

The presented results imply insights of which some were anticipated and others came by surprise. A discussion of these is divided into one section for each experiment group.

4.1 Minimax extensions

Most surprising are the results for DAB. Despite using only a single extension more than the minimax benchmark player AB, it achieved the best runtimes without compromising on decision quality and win rates. This speed up is not only caused by the dynamic board leaving out redundant recalculation of data but also by its move generator providing moves in an order that is superior to that of naive move generation. Even though this superior move generation was not designed on purpose, it leads to alpha-beta pruning setting early tight bounds, thereby early cut-offs, resulting in less states being explored as documented. Since the intent of reducing the number of explored states is to save time, generating advantageous orders without the cost of intentional sorting is especially favorable.

The deliberate move ordering of DABS, though adding computational effort to override the dynamic board inherent ordering, still achieved its target of most accurately predicting which move would be evaluated the highest at the end of minimax. This is the reason for DABS consistently exploring the least states and competing in runtime with DAB. The cost of ordering is exposed prominently on the 14×17 by DABS while exploring less states while taking longer than DAB. If the dynamic board could be upgraded to also calculate order scores more efficiently then that could lead to further performance

improvement. Moreover move lists do not necessarily need to be completely sorted, merely keeping three moves with the highest score at the front of generated lists maybe could suffice.

The move ordering of `WDABS` being less accurate than that of `ABS` shows that using window pruning changes move rankings. The difference arises through speculative pruning leading to relevant search information being cut-off whereby search quality per layer worsens as proven by `WDABS` with depth limit 8 losing against the faster `DABS` with depth limit 7. Nevertheless, as expected, when granted a big enough depth limit advantage, `WDABS` can beat players without speculative pruning because by searching less deeply, these lose search information remaining beyond their search horizons. So confining board analysis to windows around moves of interest and ignoring possible inferences from outside, despite being speculative, does not completely diminish the informational value of windowed minimax search. Like Hawaiian checkers the game of Amazons breaks into separate subgames [Hea]. But unlike Hawaiian checkers where decomposition happens after the spread of empty fields, board splits in the game of Amazons can occur during the first few rounds. Endeavors of exploiting this for state evaluation instead of pruning were successful [SM14]. Probably, the decomposition property of Hawaiian checkers endgames could be better exploited similarly than with window pruning.

Another unanticipated insight is how much the comparative performance of window pruning worsened by switching to the 14×17 board while maintaining depth limits and window dimensions. Inspired by the narrow konane boards in combinatorial game theory, it was expected that focusing on restricted areas works better when the board is a non-quadratic rectangle as interaction surfaces between portions of the board are confined like the diameter of a tube. Multiple reasons can explain why this did not happen. The tube effect cannot be recognized when transitioning from a smaller board to a bigger board that encompasses the smaller one. Generally, bigger boards take longer to decompose into subgames. Furthermore, by windows becoming smaller in relation to the complete board, window pruning cuts off bigger parts of the search tree whereby

bigger shares of potentially relevant information is lost, leading to accelerated depreciation of search quality. The last reason that implies two disadvantages at the same time is that on non-narrow boards with a bigger area, the ratio of board edge fields to enclosed fields is smaller. It implies an increased possibility of interference from outside of windows from multiple directions. It also implies that windows more often are entirely encompassed by the complete board borders and do not leap beyond them with parts of the window area. When windows encompass more fields that are possibly occupied with movable pieces, more game states have to be explored within the same search depth because of higher branching factors.

4.2 Monte Carlo tree search extensions

The clear pattern of completed iteration counts growing for both players with progressing turns arises from the number of random moves that have to be simulated between newly extended nodes and terminal states gradually decreasing. `DMCTS` did not enjoy the benefit that minimax players experienced when using the dynamic board. The only reasonable explanation for `DMCTS` completing much less iterations than `MCTS` is that the dynamic board slows down the simulation phase. In addition to missing out on the benefit of advantageous generated move orders and enabled alpha-beta cut-offs, two further causes were identified. Firstly, unlike minimax where within depth limits all children of nodes, except pruned ones, are explored one after another, Monte Carlo tree search simulation phases do not benefit from the dynamic board preparing all possible moves at all times as it only requires a single random move per layer in iterations. Secondly, since the dynamic board is more complex than the naive board implementation, copying it anew for each simulation phase takes longer than copying the naive board. The subtree inheritance extension could not have slowed down either player but as already discussed, its merit shows up only when Monte Carlo tree search plays well and accurately predicts opponent moves. The win rates for `MCTS` and `DMCTS` turning out as equal implies that in spite of collecting significantly more data in one case UCT did not manage to derive

better move decisions. Since, as already discussed, if provided with enough data UCT converges to optimal minimax play, it seems that the variance in Hawaiian checkers rollouts produces too much noise for mcts to overcome in provided time budgets. This interpretation is reinforced by the for most parts of a match modest variance in estimated best child win rates that abruptly explodes close to terminal states where simulations are very short and less entry points for noise exist. As argued in subsection 2.5.5 about limits of Monte Carlo tree search, a strong weight of noise indicates that long-term effects of Hawaiian checkers moves are small and hard to recognize.

4.3 Minimax against Monte Carlo tree search

By steadily providing `DMCTS` with exponentially growing runtime budgets the last experiment group solved the problem of Monte Carlo tree search with UCT not improving its performance despite running more iterations. The with growing `BM` sub-exponentially increasing win frequencies for `DMCTS` proved that collecting more simulation samples does indeed improve search quality although rewards grow slower than runtimes. Monte Carlo tree search performing best in the setting with the the most shallow minimax search shows that the main driver of `DABS` beating `DMCTS` is its profound search quality when searching more deeply. The results are coherent with those of other games. Usually, at least one of three different explanations can be arrived at when Monte Carlo tree search plays a game more efficiently than minimax. The two most most critical attributes of games for the relative success of Monte Carlo tree search are the lack of accurate heuristics for game state evaluation [CBSS08] and enormously high branching factors [BPW+12]; in other words the amplified presence of obstacles for minimax. Hex [AHH10], go [CMT], gomoku [TZSL16] and shogi [SHS+18] are foremost known examples of games that combine both of these attributes whereby the performance of Monte Carlo tree search for them reached state-of-the-art. It must be noted that their best Monte Carlo tree search based players usually do not rely on UCT but on more advanced deep learning based tree policies [SHS+18, SMBT21]. A third cause can be

that the benefit of moves is so prominent - and noticeably stands out from noise in the long run - that UCT focuses most of its computation budgets on exploitation and reduces exploration[Coo21]. But for games where these three characteristics are less exaggerated minimax remains preferable. Examples are the aforementioned strongest chess and checkers players [BW13, BPW+12, CBSS08]. Hawaiian checkers does not carry these properties either. Its average branching factor is even smaller than that of chess, though less studied it still provides indicators that are useful enough to base state evaluations on [Gyl76] and as demonstrated by presented results, best moves are hard to recognize.

5 Conclusion

To find out how Hawaiian checkers is played best and whether Monte Carlo tree search or minimax is more suitable for this use case, a systematic comparison of computer players based on both algorithms and introduced extensions was conducted.

The novel invented dynamic board for updating based valid move list generation and game state evaluation proved to be the most successful out of all minimax extensions. It managed to support all core components of minimax, sometimes not even by design but incidentally. The dynamic board did not manage to speed up Monte Carlo tree search simulations because these depend on other core aspects than minimax recursions. The newly designed heuristic move ordering set the earliest observed tight alpha-beta pruning bounds but could not speed up player programs that already use the dynamic board. It could be improved in future work by incorporating the computation of order scores through updating in the dynamic board. The novel invented speculative window pruning for combinatorial games that decompose into subgames cut off too many relevant search tree sections to actually improve the application of minimax search for Hawaiian checkers. However, at least some experimental results confirmed the notion that the effects of most moves are restricted by an aspect of locality. The decomposition property should be better exploited in future work by setting up data bases of subgame results and using these for partial combinatorial game-theoretic value based state evaluation. The search quality of Monte Carlo tree search with UCT was not satisfying in for competitive runtime budgets. This lead to subtree inheritance not bearing noticeable fruits. Furthermore Monte Carlo tree search mostly lost against minimax search and seems to be dependent on domain knowledge to possibly be able to com-

pete in future research. For Hawaiian checkers enough useful domain knowledge can be translated into state evaluation heuristics. Furthermore the game tree branching factor on a 8×8 board still allows for exhaustive exploration of a sufficient number of layers to inform good move decisions. In conclusion, Hawaiian checkers properties align better with suitable use cases for minimax.

Bibliography

[AHH10] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[Bjo02] Yngvi Bjornsson. *Selective Depth-First Game-Tree Search*. PhD thesis, CAN, 2002. AAINQ68547.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[Bra82] Max A Bramer. Game-playing programs: theory and practice. *ACM SIGART Bulletin*, (80):97, 1982.

[BW13] Hendrik Baier and Mark HM Winands. Monte-carlo tree search and minimax hybrids. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

[Cha10] Guillaume Maurice Jean-Bernard Chaslot Chaslot. *Monte-carlo tree search*, volume 24. Maastricht University, 2010.

[CHJH02] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[CMT] Adrien Couëtoux, Martin Müller, and Olivier Teytaud. Monte carlo tree search in go.

[Coo21] Michael Cook. Monte carlo tree search with reversibility compression. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021.

[CT02] Alice Chan and Alice Tsai. 1× n konane: a summary of results. *More Games of No Chance*, pages 331–339, 2002.

[dBP96] Arie de Bruin and Wim Pijls. Trends in game tree search. In *SOFSEM'96: Theory and Practice of Informatics*, pages 255–274, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[Eck87] Roger Eckhardt. Stan ulam, john von neumann. *Los Alamos Science*, pages 131–136, 1987.

[Ern95] Michael D Ernst. Playing konane mathematically: A combinatorial game-theoretic analysis. *UMAP Journal*, 16(2):95–121, 1995.

[FGG+73] Samuel H Fuller, John G Gaschnig, JJ Gillogly, et al. *Analysis of the alpha-beta pruning algorithm*. Department of Computer Science, Carnegie-Mellon University, 1973.

[GKS+12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[Gyl76] Joel H. Gyllenskog. Konane as a vehicle for teaching ai. *SIGART Bull.*, (56):5–6, feb 1976.

[Hea] Robert A. Hearn. *Amazons, Konane, and Cross Purposes are PSPACE-complete*, page 287–306. Mathematical Sciences Research Institute Publications. Cambridge University Press.

[Hen18] Kaitlin Hendrick. Analysis of artificial intelligence techniques for konane. *Undergraduate Thesis, Texas Christian University, Fort Worth, TX (USA)*, 2018.

[Jav] Hawaiian checkers player implementations. https://git.tu-berlin.de/shokry/hawaiian-checkers. Accessed: 08.06.2023.

[KM75] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[Kor90] Richard E Korf. Depth-limited search for real-time problem solving. *Real-Time Systems*, 2(1-2):7–24, 1990.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, pages 282–293. Springer, 2006.

[KSW06] Levente Kocsis, Csaba Szepesvari, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1:1–22, 2006.

[Mar86] T Anthony Marsland. A review of game-tree pruning. *ICGA journal*, 9(1):3–19, 1986.

[MPT22] Shiva Maharaj, Nick Polson, and Alex Turk. Chess ai: Competing paradigms for machine intelligence. *Entropy*, 24(4), 2022.

[Myc92] Jan Mycielski. Games with perfect information. *Handbook of game theory with economic applications*, 1:41–70, 1992.

[RM93] Alexander Reinefeld and T. Anthony Marsland. Heuristic search in one and two player games. 1993.

[RSS12] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. Understanding sampling style adversarial search methods. *arXiv preprint arXiv:1203.4011*, 2012.

[SBK+07] Jonathan Schaeffer, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317:1518–1522, 10 2007.

[Sch89] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[Sch11] M.P.D. Schadd. *Selective search in games of different complexity*. PhD thesis, Maastricht University, January 2011.

[ŚGSM23] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, Mar 2023.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Siz91] Walter S Sizer. Mathematical notions in preliterate societies. *The Mathematical Intelligencer*, 13:53–60, 1991.

[SM14] Jiaxing Song and Martin Müller. An enhanced solver for the game of amazons. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):16–27, 2014.

[SMBT21] Dennis J.N.J. Soemers, Vegard Mella, Cameron Browne, and Olivier Teytaud. Deep learning for general game playing with ludii and polygames. *ICGA Journal*, 43:146–161, 2021. 3.

[SP96] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 24th ACM Conference on Computer Science, CSC '96, Philadelphia, PA, USA, February 16-18, 1996*, pages 124–130. ACM, 1996.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017.

[Stoa] Ccrl 40/15 rating list april 1, 2023. https://web.archive.org/web/20230405195319/http://www.computerchess.org.uk/ccrl/4040/. Accessed: 28.04.2023.

[Stob] Ccrl 40/2 frc rating list april 6, 2023. https://web.archive.org/web/20230406194349/http://www.computerchess.org.uk/ccrl/404FRC/. Accessed: 28.04.2023.

[Stoc] Ccrl blitz rating list april 6, 2023. https://web.archive.org/web/20230406194349/http://www.computerchess.org.uk/ccrl/404/. Accessed: 28.04.2023.

[Tho05] Darby Thompson. Teaching a neural network to play konane. *Undergraduate Thesis, Bryn Mawr College, Bryn Mawr PA (USA)*, 2005.

[TZSL16] Zhentao Tang, Dongbin Zhao, Kun Shao, and Le L.V. Adp with mcts algorithm for gomoku. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 2016.

[Uit21] Jos Uiterwijk. Solving narrow konane boards. *ICGA Journal*, 43:162–183, 2021. 3.

[vN28] John v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, Dec 1928.

[Win17] Mark HM Winands. Monte-carlo tree search in board games. In *Handbook of Digital Games and Entertainment Technologies*, pages 47–76. Springer, 2017.