MTD-H: Using Conspiracy Numbers To Solve Game Trees In A More Informed Manner

J.C. Groeneveld

October 12, 2023

Abstract

Games like Chess, Go or Tic-Tac-Toe have been around for a long time. But programmable computers have only been accessible for the last 50 years. One of the areas that has been researched, is the science of rational decision making (e.g. decision theory) while other rational actors are also making decisions, and impacting the world (e.g. game theory). The pursuit of letting computers mimic rationality has also been applied to games like Chess, through solving the game's underlying game tree. In this work, we introduce MTD-H, an alteration of the MTD-bi algorithm, used for solving game trees for games like chess. We evaluate a chess engine implementing this MTD-H algorithm, and compare it to other solving methods as well.

Contents

1 Introduction

Letting computers play games like Chess has been a researched topic for many decades by now. This category of games, that chess belongs to, can be reduced to solving a game tree. First we will go over the properties of game trees, as well as examples for how one can go about solving them. We will also briefly touch on methods to perform less work to solve these game trees, touching on the concept of the minimal search tree. As will be shown, one commonly applied method for reducing the amount of work performed

searching, is using transposition tables to avoid re-searching identical positions.

Over the years many algorithms for solving these game trees have been proposed. We will touch briefly on popular game tree search methods, as well as those search methods that are most alike in implementation to MTD-H. Starting with covering the most commonly known algorithms for solving game trees: Alpha-Beta pruning and SSS*. Furthermore, we will dive into the MTD framework and its most common implementation MTD-F.

Finally, we will introduce MTD-H, an alteration of the MTD-bi algorithm. MTD-H uses conspiracy numbers to generate a probability distribution, and aims to eliminate as much of the area of that distribution with each search, instead of aiming to just reduce the search range. This new algorithm will be evaluated using the Sn0l engine[8], comparing it to different search method implementations in the same engine.

2 Background

2.1 Introducing Game Trees

Imagine you are playing Tic-Tac-Toe: Each of you take turns occupying a space on the board; this repeats until one of the players makes three in a row, or the board is full; If one of the two players occupies three spaces in a row, that player wins and the other player loses. If the board is full, without any player having three in a row, the game ends in a draw.

In the above example, certain properties about Tic-Tac-Toe can be extracted:

1. *Sequential*: Players alternate turns making a move.

2. *Perfect information*: There is no information about the game that is unknown to the players. The problem lies in finding the best line of play against an opponent that plays perfectly.

These are the two properties that are generally agreed to be part of the definition of *combinatorial games* [5, 3, 7]. Although the exact definition of a combinatorial game differs from literature to literature, one can think of combinatorial games as games like Chess, Go and Tic-Tac-Toe. There are, however, more properties that are commonly associated with combinatorial games:

1. *Zero-sum*: A game is considered to be a zero-sum game, when the total reward among all outcomes among all players is constant. Thus, the reward one player gains, must be lost from another player.

2. *Finite*: It is a common assumption that combinatorial games end after a finite number of moves.

3. *Normal* vs. *Misère*: In a *normal* game, the player that can move wins. Whereas in a misère game, the player that is unable to make the last move wins. In the case of chess, neither applies, since a game can end in a draw through stalemate.

4. *Loopy*: A game is loopy if previous game states can be revisited. Chess can be considered loopy as repeating a position once is allowed. In chess, repeating a position a third time would end the game in a draw, thus making it a different game state.

5. *Impartial* vs. *Partizan*: A game is considered impartial if both players could make the same moves in the game state, otherwise it is partizan. Chess is partizan, as the 'white' player is only allowed to move the white pieces, and for the 'black' player vice versa.

2.1.1 An Example

In Figure 1 we can see an example of a game tree, which could represent an imaginary game, with the square nodes being the first player, and the round nodes being the second player. The tree starts at the top, the root, which could represent the current position of the game: Should the player at the top choose to go left or to go right?

The values in a game tree, and Figure 1, are commonly used to represent the score that the first player will get. Since these games are zero-sum games (everything that one player gains, the other one loses, and vice versa), we only need to keep track of the score the first player gets: The first player will always try to maximize it, and the second player will try to minimize it.

In Figure 1, what should the players do in every situation? The maximizing (also known as the first) player would like to know the score at node c, and whether to go left or right. But to know c, the score of node b needs to be known. But for node b to be known, the value of node a needs to be known.

Node a is a square node: So the maximizing player is to move: It will always go the way that will score the most. The maximizing player will

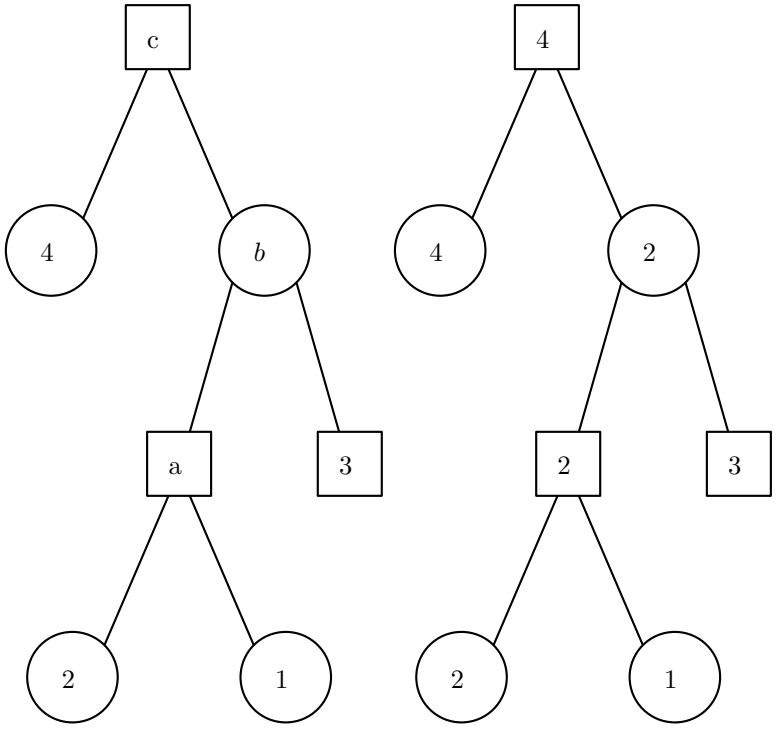

Figure 1: An example of a game tree, both unsolved and solved.

always choose the node with the value 2 in this case. So whenever the game ends up at node a, the game will eventually end up at the node with value 2. This means we can assign a score of 2 to that node: If the game reaches that position, the maximizing player will always score 2 in the end.

Now that we know the value of a, we can calculate the value of b. This node is represented by a round node: It is the minimizing player's turn to choose. The minimizing player, knows that if he were to choose to go to a, the first (maximizing) player would end up with a score of 2. Since otherwise the game would end with a score of 3, this is the best option for the minimizing player. The minimizing player would always choose the node with score 2 in that position, so whenever that position is reached the game will end up with a score of 2.

Finally, we can calculate c. Here the maximizing player will choose the node with score 4, over the node with score $b = 2$.

2.1.2 Dealing With Duplicates In The Game Tree

Imagine you are playing Tic-Tac-Toe, and you reach the position shown in Figure 2. The game will play out the same way, regardless of which 'X' was played first. Whether the game played out 'Top-Left Bottom-Left Center' or 'Center Bottom-Left Top-Left', the game tree to solve from this point on is the same.

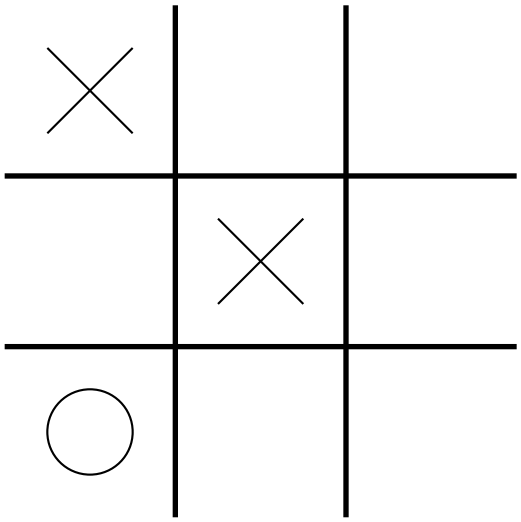

Figure 2: An example position of the game of Tic-Tac-Toe.

Now imagine you are currently deciding your Tic-Tac-Toe move at an empty board. Different sequences of moves can lead to the same position as shown in Figure 3, with identical game trees.

It is not desirable to re-search identical game states, as the evaluations of those identical states would also be identical. If those outcomes could be stored, and efficiently retrieved later, this extra work could be skipped. Many programs solving game trees employ a database to store states already encountered. This is usually done by storing a hash of the game state and the game state evaluation as a (key, value) pair in a hash table. This hash table is often referred to as a 'Transposition Table', for when a move transposes a position into an already calculated position.

The hashing associated with hash tables can come at a price. A bad hashing algorithm may cause many hash collisions, while a better algorithm may be computationally expensive. In games like chess or checkers, this is remedied through the use of Zobrist hashing [23], a form of tabulation

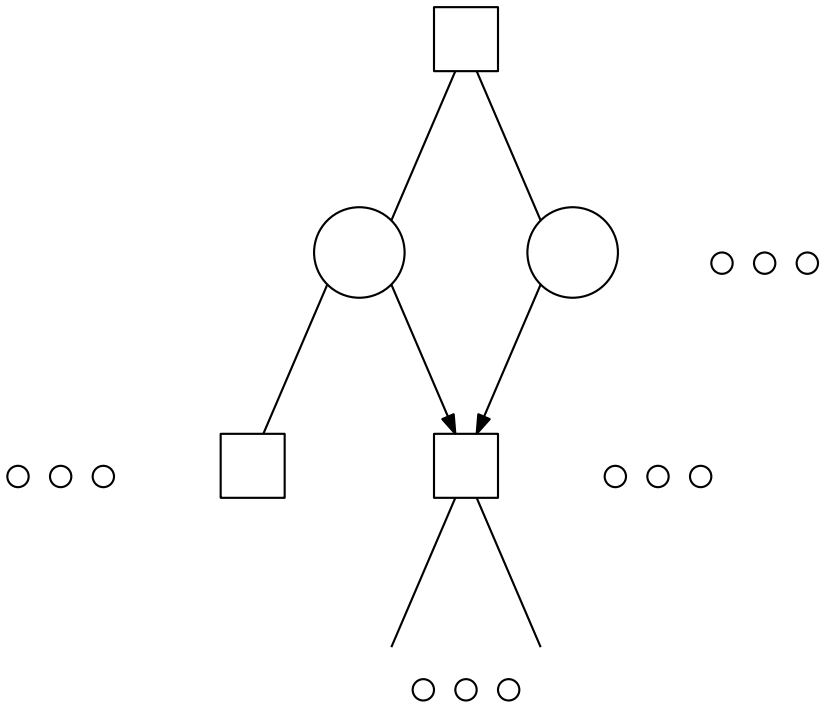

Figure 3: A game where different sequences of moves may lead to the same position.

hashing.

In e.g. chess or checkers, the game state can be represented as a composition of pieces. The general idea is that, in those games, the hash of a game state is the composition of the hashes of its pieces (or, for games without pieces, more abstractly, some chosen components). So a piece, of a particular colour, being on a particular square constitutes a unique hash, given that enough bits are allocated to represent all possible combinations. It uses the XOR operation to compose the final hash of a game state as a composition of the hashes of those pieces. A move can consequently be represented as the difference between the presence of pieces. Meaning that the change of the final hash of the game state can be portrayed as the difference between the parts it is composed of. The presence of their hashes as part of the final hash of the game state, can thus be calculated through simply XORing the hashes of those pieces. Which is the same as XORing the old hash with the hash of the move to create the new hash. This grants significant speedups compared to fully recalculating the hash for every position,

as well as keeping the hash collisions at an acceptable level [23].

The data stored in the Transposition Table, corresponding to an already searched position, can be more than just the calculated value of that game state. It could also store lower- or upperbounds on the value, or even more, such as at what depth the position to, or the best move from that position (useful for deciding what move to order first when searching at a higher depth).

2.2 Alpha-Beta pruning

It is often the case that it is not necessary to calculate what move to make for every position, but only for the current position: Just solve the root of the current game tree. This allows us, as we shall see, to skip calculating the score for positions that we can guarantee will never be reached anyway.

One of the first, and most popular, algorithms developed for solving this problem is α-β search. This α-β search evaluates the tree from left to right, while keeping track of a narrowing window of allowable values.

The idea is that when a move becomes "good enough" for a player, we can stop looking whether the move is even better: Because the other player is able to disallow a move "that good", since it already can force a line of play which is calculated to not be "that good". As we calculate more and more of the tree, these thresholds for whether a move is good enough will narrow, allowing us to skip more and more calculations.

There are two thresholds: α and β:

1. When a move is guaranteed to be less than α, we can skip continuing to evaluate the value of that move.

2. When a move is guaranteed to be more than β, we can skip continuing to evaluate the value of that move.

An example can be seen in Figure 4: After evaluating the left-most node with value 4, and continuing onto the node to its right, that lower threshold, α is at least 4. So once we know that that node has a value of at most 2, we can stop evaluating it. It will never be 4 or more, so the maximizing player will not go there, regardless of what its value is precisely.

2.3 Minimal Search Tree

The right tree in Figure 4 is imperfectly ordered in the sense that the best move is not always first, but will be solved by Alpha-Beta Search with less

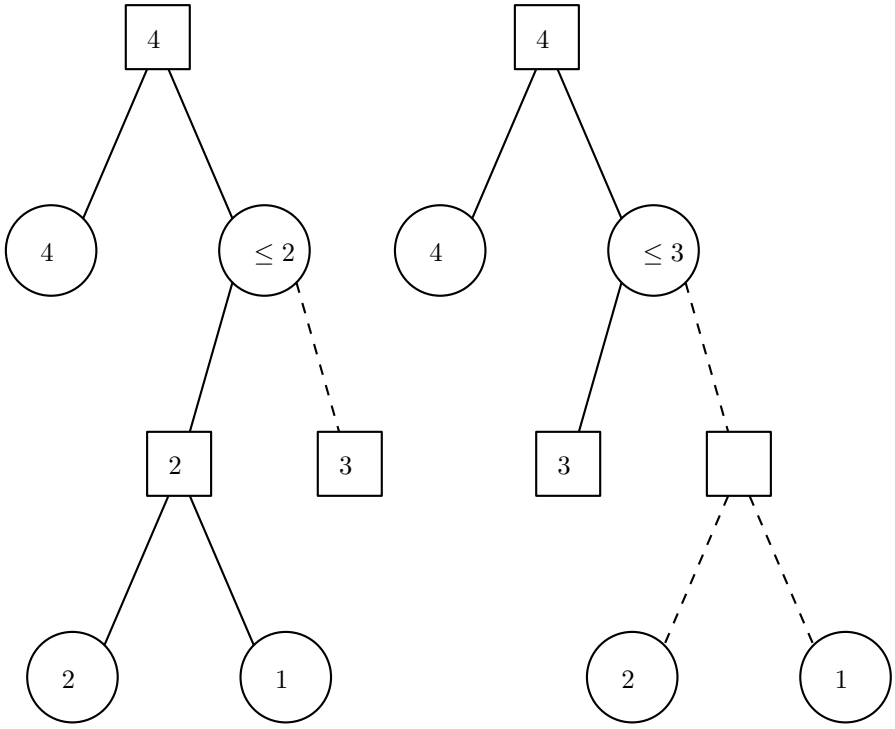

Figure 4: The same tree, with perfect- and imperfect ordering, showing which branches are pruned by Alpha-Beta search.

work. A lower bound on what must at least be searched, is shown by Knuth et al. [9]
This changes what the "perfect move order" is: Instead of always choosing the move that will turn out to have the highest evaluation, it changes to the move that still leads to a cutoff with minimal work as possible. An example for the game Go: Suppose, if you cannot force an opponent's group (a connected collection of stones) to die given a certain move you are considering, that whole branch will be cut off (since, supposedly, you already found another move you can make that will kill the opponent's group, and which thus has a lot move favourable evaluation). The best move order within that branch you are still calculating is to go for the opponent's move that will most likely allow that group to live, with the smallest amount of work, thus allowing you to cut off that branch. More elaboration regarding this subject is done by Plaat et al. [13].

2.4 SSS*

Another algorithm, like α-β pruning, is SSS* [15, 19]. SSS* is designed to work with game trees where the root is a max node. Every node that is expanded by SSS*, would also be expanded by α-β pruning: It is guaranteed to not need to evaluate nodes, which α-β pruning also does not evaluate.

Instead of evaluating the tree from left-to-right, the idea is to evaluate the "most promising" candidate first. "Most promising", does in this case denote the node with the highest value. Nodes with an unknown value have a value of $+\infty$.

2.5 Search With Small α-β Windows: The MTD Framework

We have looked at α-β pruning. What would happen if we were to do an α-β search with an already chosen lower bound (α) and upper bound (β) which are not at infinity? It would allow us to prune more branches, but would we still be able to make the same conclusions from a search like this?

There are three different cases to look at:

1. The actual solution to the game tree is lower than α.
2. The actual solution to the game tree is higher than β.
3. The solution is between α and β.

Let us first consider the case where the solution is lower than α. If, for a branch starting at a max node, its solution would be lower than alpha, then, the maximum value of its children is less than α. So, its eventually calculated value will be a value less than α. This is after having calculated the value of all of its children. If, for a branch starting at a min node, its solution would be lower than alpha, then, the minimum value of its children is less than α. When calculating the value of its children from left to right, eventually a child will have a value lower v than α. The calculation of the value of the root of that branch will stop, and its value is simply known to be at most v. Thus, an exact evaluation is not known. In the case that the solution turns out to be higher than β, the same logic applies as when the solution is lower than α, but with 'min' and 'max', 'lower' and 'higher', 'less' and 'more' swapped.

In the last case, just as with normal α-β search, when starting a search on a branch with its solution in between the α and the β, the correct solution will be found. This follows from the fact that, if the game tree still contains the leaf node which leads to the evaluation of the whole game tree, α-β search will still correctly solve the game tree.

2.5.1 Searching with windows of minimal width: MTD

Performing an α-β search on a game tree with a 'null-size' window, a window with $\alpha + 1 = \beta$, was introduced by Judea Pearl [12], and subsequently called "Pearl's Test procedure".

Now, 'MT' is a "Memory-enhanced version of Pearl's Test procedure" [14]. It combines performing 'Pearl's Test' with the use of transposition tables, and iterative deepening. 'MTD' stands for 'Memory-enhanced Test Driver': The framework of repeatedly performing MT.

When introducing MTD, Plaat et al. [14] show that MTD-F, an instance of an MTD algorithm, delivers competitive performance for solving game trees, outperforming NegaScout, which was considered to be the previous best α-β implementation. They also show that the MTD framework can behave equivalently to SSS* in terms of search order and nodes searched [15].

2.5.2 MTD-F and MTD-Bi

Two algorithms that follow the MTD framework are MTD-F and MTD-Bi.

MTD-F works as follows:

1. Start a search at point 'h'. (A heuristic starting point, for chess engines usually the solution to the tree at a lower depth).
2. As discussed, this will either return a lower-, or upperbound.
3. If a lower- or upperbound was found, perform a search again.
4. If lowerbound $\geq$ upperbound, then return the found value

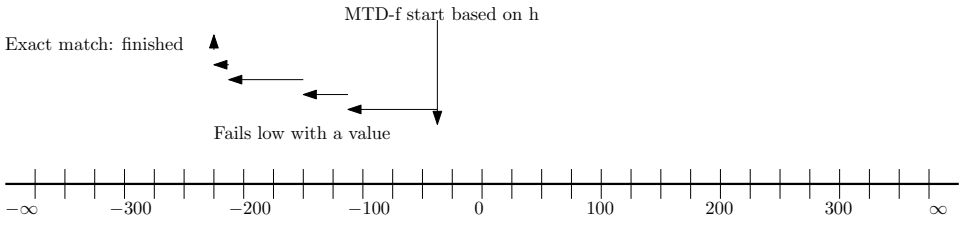

Figure 5: MTD-F performing repeated searches with different window values until it converges on the answer

MTD-Bi works by using the tests as a binary search over the range of possible values. As is illustrated in Figure 6.

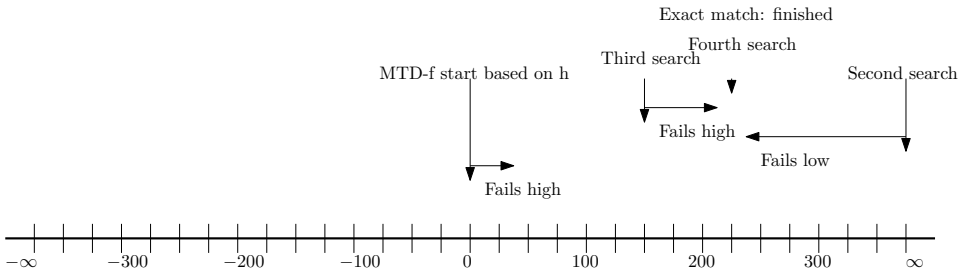

Figure 6: How MTD-Bi performs tests with minimal window size repeatedly

2.6 Conspiracy Numbers

Conspiracy numbers are a concept introduced by McAllester et al. in 1988 [11]. It gives an indication of how susceptible the final evaluation of the game tree is, with respect to changing the value of its leaf nodes. The lower the conspiracy number, the less leaf nodes need to be changed to change the outcome of the game tree.

Or explained more formally, $\mathcal{C}$ is a set of conspirators which, when all changed together, can change the value of game tree T to value V. A conspirator is a leaf node of subtree T, which is not a terminal node of the game. With Δ-needed(i, T, V) being the number of conspirators required to change the minimax of node i in tree T to value V, representing the cardinality of its corresponding set $\mathcal{C}$. Or in short: The conspiracy number. Furthermore, $\uparrow$ needed(i, T, V) represents the number of conspirators to get at least V. And $\downarrow$ needed(i, T, V) represents the number needed to get at least evaluation V. Both $\uparrow$ needed(i, T, V) and $\downarrow$ needed(i, T, V) are shown to be monotonic by McAllester.

The idea behind the usefulness of Conspiracy numbers for inspiring this research is as follows: When using iterative deepening, and re-searching from a lower depth, the conspiracy number for achieving a given value V may correlate with the probability that, at a higher depth, the minimax value will end up at that value V.

2.7 Proof Number Search (PNS)

Another application of conspiracy numbers, is their use in Proof Number Search [4, 20], and variants [22, 18, 17]. This search method is useful for searching game trees with imbalanced branching factors: Game trees where there are branches that contain way more nodes than other branches. An excellent explanation of Proof Number Search is given by Van Den Herik et

al. in 'Proof-number Search and its Variants' [20], in section 2.1.

A quick summary of the main idea of Proof Number Search: When evaluating a multivalued game tree, one can split up the values in the tree to "at least V" and "less than V", in the style of Pearl's Test Procedure. Changing the tree into an AND-OR tree.

1. With all terminal nodes with at least value V into terminal nodes with value 1.

2. All terminal nodes with less than value V into terminal nodes with value 0.

3. All max nodes into OR nodes.

4. All min nodes into AND nodes.

For each subtree, it keeps track of the number of the number of terminal nodes that need to be examined to prove the value of that subtree to be 1 (the proof number), or 0 (the disproof number). The proof number of an OR node is the minimum of the proof number of its children. And, the disproof number is the sum of the disproof number of the children of that OR node. For AND nodes, it is vice versa. An unevaluated terminal node has $(\text{proof}, \text{disproof})$ pair $(1, 1)$.

This way, in unbalanced trees, it can keep on trying to prove the value of the tree (it having value 1), or disprove the value (it having value 0), by continuing to shrink the proof number in OR nodes, through minimizing the proof number of the child with the smallest proof number. And the same applies for disproof numbers in AND nodes.

Proof Number Search and all its variants are designed to work only on AND-OR trees. But, they can be expanded to (multivalued) game trees by repeatedly performing Proof Number Search on game trees which are split on V into AND-OR trees.

An exception to this procedure only working on AND-OR trees, is the work by Saffidine et al. [17]. It replaces proof- and disproof numbers with 'effort numbers':

1. $G(n, o)$ is the number of node expansions n required to prove that the outcome of a (sub)tree is at least value o.

2. $S(n, o)$ is the number of node expansions n required to prove that the outcome of a (sub)tree is at most value o.

It is an expansion of PNS to multivalued game trees using these effort number. It works by establishing an "attracting outcome" for a (sub)tree, which is an outcome o with $\min(G(n,o) + S(n,o))$ where $G(n,o) > 0$ for max nodes, and $S(n,o) > 0$ for min nodes. As well establishing a "detracting outcome", which is the outcome o that is just worse (for max nodes), or just better (for min nodes).

Given that the root is a max node, it continues to run in the same style as Proof Number Search, trying to prove its (changing) *attracting outcome* at max nodes. And prove its *detracting outcome* at min nodes.

3 Relevance

Monte Carlo Tree Search is a more recent development when it comes to searching game trees [5, 6]. It has garnered a lot of interest compared to the more classical approaches due to its good performance in games like Go and Chess. Unlike algorithms like Alpha-Beta and SSS* search, it does not aim to exhaustively solve a game tree, and is thus usable for estimating the best move in computationally intractably large game trees.

So, although Monte Carlo Tree Search attempts to solve a different problem than the algorithms of the MTD framework (such as MTD-H), it is not wholly uninteresting. By foregoing the requirement to search the game tree at equal depth throughout the whole tree, it gains an edge on classical engines [5]. It works in four phases [6]: Selection, Expansion, Simulation, Backpropagation.

3.1 Selection

A separate tree of visited nodes from the game tree is kept. For each node, a number of things are kept track off: What game state it is associated with (Which node it is in the game tree), which action (from the previous state) led to it, its average reward, and the number of times it was visited. For each node a score can be calculated based on its average reward, and the number of times it was visited. This allows for a scoring with an exploitation/exploration trade-off [5]. Starting from the top node, each time the child node with the highest score is picked. This tree is traversed until a node is visited which has a game state which has unvisited children.

3.2 Expansion

When a node is encountered with a game state that has actions that are not yet stored in the tree from the selection process, a new node is created and appended to the tree.

3.3 Simulation

From the selected node, and corresponding game state, the following happens: According to some "default policy", which is a simplified, dumbed down way to simulate the game playing out, actions are chosen until the game terminates with a resulting reward. For example, for a game like chess this would be $0, \frac{1}{2}, 1$ for a loss, draw, and win respectively.

3.4 Backpropagation

For the new node, and each of its parent nodes all the way to the root, the number of times that node is visited, as well as their average reward is updated. This results in a new score for those nodes, which helps influence the selection phase for the next iteration of the search.

This search is continued until the computational budget is expended. After the search the node with the best exploitation score is selected to be played.

4 Methods

4.1 Searching

Multiple search methods are compared. All the search methods are implemented in the Sn0l engine.

First off, as a baseline α-β search is implemented. Furthermore, MTD-F and MTD-Bi search are implemented, with- and without conspiracy numbers. Although MTD-F and MTD-Bi don't make use of the conspiracy numbers, the inclusion of the search with conspiracy numbers is done to get an impression for the performance impact that gathering conspiracy numbers would have. The last search algorithm that is tested, is MTD-H. MTD-F and MTD-Bi use the same MTD framework implementation, whereas MTD-H and the conspiracy search variants of MTD-F and MTD-Bi use a modified MTD framework implementation which accommodates the use of conspiracy numbers.

4.1.1 MTD-F and MTD-Bi Search Without Conspiracy Search

The implementation for MTD-F and MTD-Bi is identical, except for their respective step functions: The step functions are the functions that dictate the search strategy, where to search next.

The Sn0l-specific implementation for the MTD-F step function can be specified in pseudocode that is derived from the original Rust code as follows:

```
function determine_mtdf_step(last_test_value: BoardEvaluation,
    lowerbound: BoardEvaluation, upperbound: BoardEvaluation) ->
    BoardEvaluation {
    // Returns a new BoardEvaluation to test at.

    if last_test_value >= upperbound {
        let mut new_value = min(upperbound, last_test_value -
    MTDF_STEP_SIZE);
        if new_value <= lowerbound {
            new_value = avg_bounds(lowerbound, upperbound);
        }

        return new_value;
    } else if last_test_value <= lowerbound {
        let mut new_value = max(lowerbound, last_test_value +
    MTDF_STEP_SIZE);
        if new_value >= upperbound {
            new_value = avg_bounds(lowerbound, upperbound);
        }

        return new_value;
    } else {
        panic!("MTD-F boundary calculations: last_test_value {
    last_test_value} wasn't lower than lowerbound {lowerbound}, or
    higher than upperbound {upperbound}");
    }
}
```

Listing 1: The MTD-F step function

A couple of things to note regarding this code snippet:

1. The type `BoardEvaluation` represents the range of values that could be achieved in the zero-sum game, in this case chess. It must be ordered, and must allow for simple arithmetic to accommodate taking averages and moving in steps. In the case of the Sn0l engine, the ordering is `Blackmate(x)`, `Centipawns(x)`, `Whitemate(x)`. With `Blackmate(0)` being the lowest value, meaning that black has mated white. The `Centipawns` represents the score difference of chess material, with a positive score meaning white is ahead in material. One Centipawn is meant to approximately represent a material difference of one hundredth of a pawn.

2. In the case of `if last_test_value >= upperbound`, the last test value is above what is now the new upperbound. So the result from the last test must have been below that last test value. That means we want to move lower.

3. In the case of `if new_value <= lowerbound`, the new test value is lower than the lowerbound. This means we would overshoot and search in a range of values of which we already know that the answer isn't in that range. So take the average of the bounds, and stay in between the bounds of possible values.

4. This algorithm is symmetric in the sign of the `BoardEvaluation`, so the same logic applies for moving the `BoardEvaluation` in the other direction.

5. The last clause, where the program panics, is if the new upper- or lowerbound did still allow for testing the last tested value, which should be impossible.

6. One thing to note is that in this implementation the step size is calculated from the last tested value, instead of the newly established upper- or lowerbound.

Now the step function implementation used for the MTD-Bi algorithm is a lot simpler:

```
function determine_mtdbi_step(last_test_value: BoardEvaluation,
    lowerbound: BoardEvaluation, upperbound: BoardEvaluation) ->
    BoardEvaluation {
    // Returns a new BoardEvaluation to test at.
    return avg_bounds(lowerbound, upperbound);
}
```

Listing 2: The MTD-Bi step function

One last implementation choice, which may have a big impact on the performance of MTD-Bi (and MTD-F to a degree), is how the "averages", and step sizes are calculated when the upper- or lowerbound contains a checkmate value. When taking an average of a centipawn value, and a checkmate value, an impossibly low or high centipawn value (depending on the mate value) is chosen as test value. This is meant to cut off either all centipawn values or all mate values from the search range. Secondly, when moving with a step size from a checkmate value, that checkmate value is returned unaltered.

4.1.2 The MTD Framework

A notable implementation detail for the MTD framework used in the Sn0l engine, is how to handle search instability. Search instability is the phenomenon where repeated MT searches will not settle onto an exact value. In this case MT searches keep returning upper- and lowerbounds that lay outside the previously established range of possible answers. It is a result of each accessing the game tree in a different order, and consequently updating the transposition table in such a manner that the final answer changes. This can happen when a particular that occurs in the game tree occurs that multiple depths, and only the evaluation for the highest depth is kept. Although some literature exists regarding on how to deal with search instability [21, 1], it is mainly contained to forum posts [2].

In the Sn0l engine, the implementation comes down to the following:

```
if unstable_search_counter > 3 {
    if result_is_lowerbound && white_to_play {
        return result;
    }
    if result_is_upperbound && black_to_play {
        return result;
    }
}

if unstable_search_counter > 6 {
    return result;
}
```

Listing 3: Code for handling search instability

First off, the unstable search gets some iterations time to still settle into an exact answer, which is preferable. After that, an answer that has a preferable bound for the player that has to make the move, suffices. This prevents blundering and allows for playing an at least near-optimal move, even though it is not guaranteed to be the absolute best move. Lastly, if the search only keeps on producing unfavourable bounds (which happens), it will return the last result.

4.1.3 Conspiracy Numbers

MTD-H, and the MTD-F and MTD-Bi implementations that collect conspiracy numbers for performance measurements, use a modified version of the MT procedure specified by Plaat et al. [14]

The way that conspiracy numbers are collected in the Sn0l engine is through the use of a data structure that, for any node in the game tree, can

store the conspiracy numbers for that node. the data structure must obey to the following restrictions:

1. It must allow creating a new instance of itself through the method `from_leaf(value: BoardEvaluation)`, which creates a new instance of that data structure that stores that one conspirator.

2. It must allow for creating a new instance for terminal nodes, using `from_terminal_node(value: BoardEvaluation)`, which stores that that value is impassable, whether you want to change the value upwards or downwards from there.

3. It needs to implement `merge_max_node_children(self, other)`. This must allow for merging the stored conspirators of the children nodes in a way that conforms with the way conspiracy numbers are merged. This implies that `merge_max_node_children(self, other)` needs to be associative and commutative, since it needs to be impervious to move ordering. The rules for how conspiracy numbers can be constructed for a max node is described in the work by McAllester [11], but the merging strategy specific to Sn0l will be discussed later.

4. It needs to implement `merge_min_node_children(self, other)`. The same rules as those for `merge_max_node_children` apply.

5. A last requirement, that is not needed for collecting the conspiracy numbers, is a way to extract the conspiracy numbers for any given `BoardEvaluation`.

The data structure used in the Sn0l engine consists of 'buckets' that represent how much the conspiracy number increases to go from the start of that bucket to the end of that bucket. The data structure is split up into two arrays. One array is for determining the $\uparrow \text{needed}(j, T, V)$. Every item in this array represents how much the conspiracy number marginally increases to go from that bucket's lowerbound (start) to that bucket's upperbound (end).

The other array is for determining the $\downarrow \text{needed}(j, T, V)$. Here the start of each bucket is its upperbound, and the end of each bucket is its lowerbound.

In figure 7, each item in the arrays represents the marginal contribution of the amount of conspiracy numbers that corresponding range of `BoardEvaluation` contributes. Here the boundaries are represented in centipawns.

The Up Buckets

0	0	4	12	54	8	17

... −50 −30 −10 10 30 50 ...
Boundary values

The Down Buckets

12	3	2	0	0	0	0

... −50 −30 −10 10 30 50 ...
Boundary values

Figure 7: The datastructure used for storing the conspiracy numbers.

4.1.4 Calculating the Conspiracy Numbers for Non-Leaf Nodes

For a max node i, with a set of child nodes $S(i)$, the following equations hold, as established by McAllester [11]:

$$\uparrow \text{needed}(i, T, V) = \min_{j \in S(i)} \uparrow \text{needed}(j, T, V)$$

$$\downarrow \text{needed}(i, T, V) = \sum_{j \in S(i)} \downarrow \text{needed}(j, T, V)$$

And for a min node, the following hold:

$$\uparrow \text{needed}(i, T, V) = \sum_{j \in S(i)} \uparrow \text{needed}(j, T, V)$$

$$\downarrow \text{needed}(i, T, V) = \min_{j \in S(i)} \downarrow \text{needed}(j, T, V)$$

So, when constructing the array of conspiracy numbers for max and min nodes, one can turn the arrays of marginal conspiracy numbers into arrays of the cumulative conspiracy numbers to get past that bucket. Then simply apply the equations provided above, and get the new cumulative conspiracy numbers. Some leaf nodes are also terminal nodes, meaning it is impossible to get past. This can be handled by setting the conspiracy value to $+\inf$ for that `BoardEvaluation`, making it impossible to get past that node.

The way that the transposition table ties into the collection of conspiracy numbers needs some consideration. Storing conspiracy numbers in the

transposition is not doable. Usually, only a few bytes worth of memory are stored per transposition entry. Adding conspiracy numbers to each entry as well, would increase the memory footprint of the transposition table at least an order of magnitude. In a world where transposition tables are already memory-constrained, this is not a desirable behaviour.

The crux of the problem is in the fact that we are trying to solve a directed acyclic graph by treating it as a game tree. So, how should we count the conspiracy numbers when a node is already in the transposition table? A solution is to pretend that a node from the transposition table has no conspirators. The conspirators were already accounted for once, when that node was put into the transposition table the first time. This allows us to not add any more memory pressure on the transposition table, and still allows for usable conspiracy numbers.

4.1.5 MTD-H Search

MTD-H is an alteration of MTD-Bi. It incorporates the idea that, while MTD-bi does a binary search over the possible outcome values, there is more information to be had about what the final outcome of the search will be. By approximating the final (and deterministic) answer with a probability distribution based on heuristics, an example is shown in Figure 8, the binary search of MTD-bi can be sped up. Instead of aiming to halve the search space with each search (as MTD-bi does), we can aim to reduce the area under the probability distribution as fast as possible.

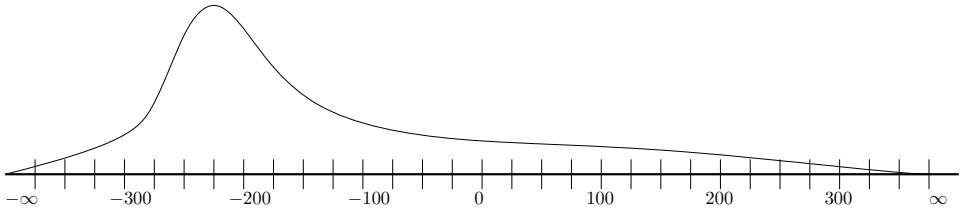

Figure 8: A range of possible outcomes, accompanied by a probability distribution.

After having performed a search, new information comes available, e.g. a new lower- or upperbound, or updated conspiracy numbers. An example of this can be seen in Figure 9.

Now, MTD-H can be split into two independent parts. Firstly, a part that calculates the estimated probability distribution from the available data. Many implementations can exist for this. Secondly, the mechanism

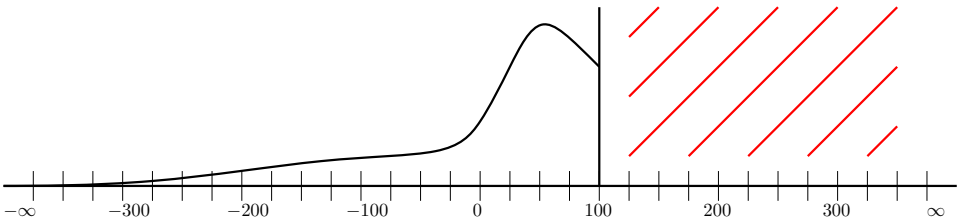

Figure 9: A new probability distribution after having performed a search.

that chooses the value which the next MT-search is performed with.

4.1.6 MTD-H: Calculating the Probability Distribution

The way that the probability distribution is made, is as follows: The conspiracy numbers from a lower depth are used to give a static estimate of the probability distribution for the current depth that we are now attempting to search. Then, as the search range is reduced, the probabilities corresponding to those dropped values are dropped as well, and the area of the probability distribution is readjusted back to one.

So, how to construct that static probability distribution? Firstly, assumptions are made. In reality, the conspiracy number represents the cardinality of the smallest set of conspirators. But now, we assume that there exists only one set of conspirators of that cardinality. Another assumption is that every conspirator has an equal probability of stopping the `BoardEvaluation` of going past that value. This ignores the fact that some leaf nodes may occur more frequently in the directed acyclic graph. Lastly, the probability of any conspirator stopping the `BoardEvaluation` is independent to that of all other conspirators.

The idea is that, at a higher depth, for the `BoardEvaluation` to change to the value of one of the leaf nodes, it must get past all the other leaf nodes that have evaluations between that of the current evaluation and the evaluation of that leaf node. Given the assumptions, we can model the probability that the final new evaluation ends up at a bucket containing x leaf nodes, which is passed y conspirators already:

```
1 pub fn bucket_probability_down(x, y) {
2     if x == +infinity || is_last_bucket {
3         // Unable to pass this bucket so all remaining probability
      gets put in here.
4         // Each conspirator has probability p of letting the final
      evaluation get passed it.
5         // p^y represents the probability that the final evaluation
      got passed all previous conspirators.
```

```
6           // Since passing this bucket is impossible, ending up at this
    bucket, given that we arrive here, has probability 1.
7           return weight_side_down * 1 * p^y + c
8       }
9       else {
10          // There are x conspirators in this bucket.
11          // Each conspirator has probability p of letting the final
    evaluation get passed it.
12          // 1 - p^x represents the probability that the final
    evaluation does not get passed all of its conspirators.
13          return weight_side_down * (1 - p^x) * p^y + c
14      }
15  }
```

Listing 4: The probability calculation for ending up in a particular bucket, with p being the probability of passing a conspirator.

In the above code example, there are still two unexplained variables: weight_side_down and c. The probability of changing the `BoardEvaluation` downwards or upwards is not assumed to be equal, so each side has their own probability. Furthermore, since there is still a probability of ending up at a bucket with no conspirators, a constant c is added to the probability of each bucket.

The calculations for changing the probability downwards are the same as those for changing it upwards. The probability for changing downwards towards a bucket is added to the probability to change upwards towards it. Finally, since the probabilities for each bucket are now known, the probability distribution is normalized so that its area adds up to one.

4.2 Auxilliary Implementation Details

4.2.1 Position Scoring

In the Sn0l engine, a leaf node is evaluated based on a piece-square tables, or if the game is over, it is evaluated by the game result. The piece-square tables assign a material score to each piece depending on what square they are standing on, and what color they belong to.

A common implementation for chess engines is the use of quiescence search in evaluating the leaf nodes. Quiescence search assigns the value of a leaf node by continuing to evaluate specific lines of play further. Commonly those are the lines of play that only consist of captures or checks. As an example, this allows for a leaf node evaluation to account for the fact that the queen, that just captured a knight, gets captured in the next turn. As a consequence, the evaluations of leaf nodes become more accurate end less sensible to big swings in evaluation as the search depth increases. Al-

though the Sn0l engine has support for quiescence search for the α-βsearch algorithm, it is not implemented for the MT-search procedures.

4.2.2 Move Ordering

Both α-β search and MT-search implementations have the same position scoring and move ordering. Moves are ordered by captured first, followed by non-captures. The captures are ordered in descending order on `piece_value(target_piece) - piece_value(source_piece)`. With `piece_value` representing the general valuation of that type of chess piece. The non-capture moves are ordered in descending order based on the difference in score in the piece-square table.

4.3 Test Setup

To compare the performance of the multiple search algorithms, the 'Win at Chess' [16] dataset is used. It is an old dataset from 1958, This dataset consists of three hundred chess puzzles. The parameters of MTD-H were optimized on the first 250 problems.

The algorithms were compared on the full set of three hundred problems, as well as the last fifty problems separately.

For recreating the test results, an analysis program can be compiled from `https://github.com/nielsgroen/sn0l`[8]. It supports performing the same analyses as were performed for this research, among other analyses which were not performed due to compute time, and mainly storage constraints. It supports match playing as well as analysing on the Lichess Opening Database[10].

The algorithm setups tested are as follows:

```
1 cargo run --bin store_analysis --release -- -a mtdh-iterative-
    deepening -s 8 --db-path sqlite://win_at_chess.db -n 101 -b 20
2 cargo run --bin store_analysis --release -- -a mtd-bi-iterative-
    deepening-conspiracy -s 8 --db-path sqlite://win_at_chess.db -n
    101 -b 20
3 cargo run --bin store_analysis --release -- -a mtdf-iterative-
    deepening-conspiracy -s 8 --db-path sqlite://win_at_chess.db -n
    101 -b 20
4 cargo run --bin store_analysis --release -- -a mtd-bi-iterative-
    deepening-conspiracy -s 8 --db-path sqlite://win_at_chess.db -n 51
     -b 20
5 cargo run --bin store_analysis --release -- -a mtdf-iterative-
    deepening-conspiracy -s 8 --db-path sqlite://win_at_chess.db -n 51
     -b 20
6 cargo run --bin store_analysis --release -- -a mtd-bi-iterative-
    deepening -s 8 --db-path sqlite://win_at_chess.db
7 cargo run --bin store_analysis --release -- -a mtdf-iterative-
    deepening -s 8 --db-path sqlite://win_at_chess.db
```

```
cargo run --bin store_analysis --release -- -a alpha-beta-iterative-
    deepening -s 8 --db-path sqlite://win_at_chess.db
```

Listing 5: The shell commands for compiling, and running the test setups.

A full list of possible options can be found in Appendix B.

The MTD-F and MTD-Bi implementations that do not use conspiracy search, are separately implemented from their respective counterparts with conspiracy search. This is done to create a more representative performance impact that conspiracy search may have.

5 Results

To determine the effectiveness of MTD-H, it is compared to a baseline α-β search, as well as MTD-F and MTD-Bi. To give an indication of the performance impact that collecting the conspiracy numbers would have, multiple variations of MTD-F and MTD-Bi with conspiracy number collection are compared as well. The algorithms are compared against on the whole set of 300 positions, as well as the last 50 positions, which were not used to tune the parameters for MTD-H.

The results are summarized in the coming figures:

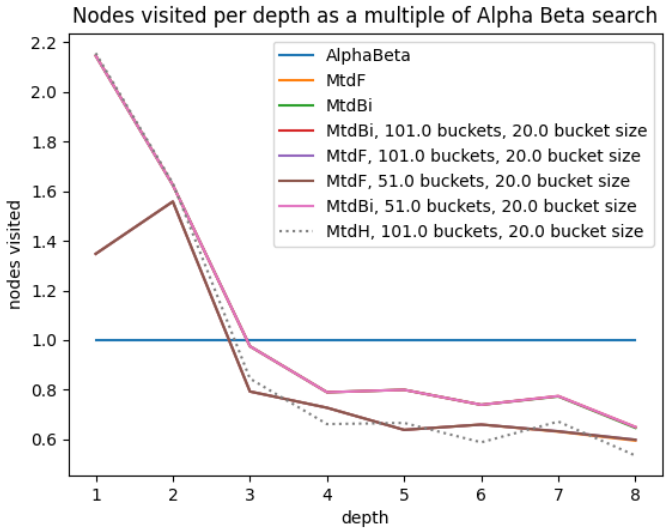

Figure 10: The number of nodes visited by each algorithm as search depth progresses.

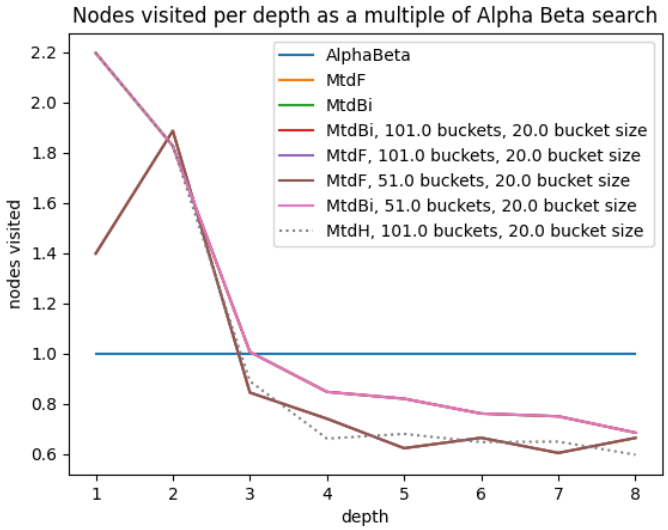

Figure 11: The number of nodes visited by each algorithm on the last 50 positions.

The MTD-H algorithm stores the conspiracy numbers in a bucket of a width of 20 centipawns, and contains one hundred buckets, allowing for collecting centipawns on a range of -1010 centipawns up to 1010 centipawns. When comparing MTD-H with MTD-F, the number of nodes visited is comparable, alternating at odd- and even depths with which algorithm searches the least number of nodes. When evaluated separately on the last 50 positions of the Win at Chess dataset, the behaviour is not much different.

In Figure 13 we can see MTD-H and the conspiracy search variants trailing their non-conspiracy search counterparts. One thing to note is that MTD-H clearly outperforms its MTD-F and MTD-Bi counterparts that have as much memory dedicated to storing conspiracy numbers as the tested MTD-H variant does.

While MTD-H is competitive when it comes to number of nodes visited. When looking at time taken, the conspiracy search variants underperformed the variants that do not collect conspiracy numbers. All conspiracy search variants use the same Memory-enhanced Test. Disregarding the performance impact due to their difference in their transposition table access patterns, and external influences such as background processes running during testing, their performance should be equal when they use the same amount of

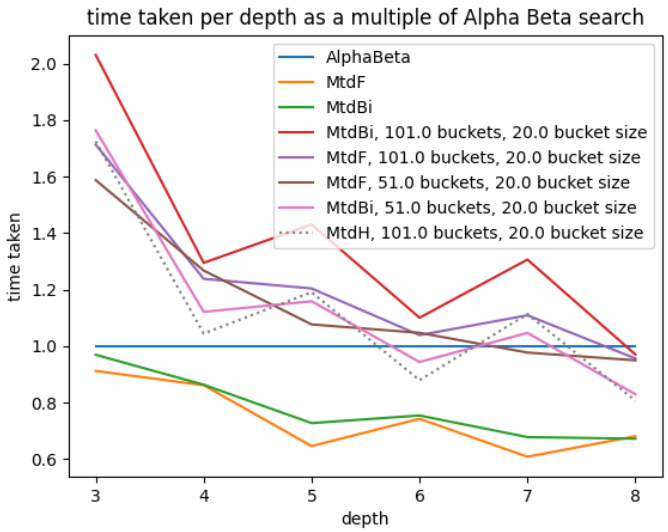

Figure 12: The time taken by each algorithm to complete calculating the positions as search depth progresses.

memory for storing the conspiracy numbers. The nodes per second for each algorithm run are displayed in Figure 14, and for the last 50 positions it is shown in Figure 15.

One last metric to compare the search algorithms on, is the number of Memory-enhanced Tests they do. The α-β search is exempt from this, as it does not work perform repeated MT searches, but instead narrows its search window as it searches. The results are shown in Figures 16 and 17.[1]

When looking at the last 50 test positions, MTD-Bi performs less MT searches than MTD-F and MTD-H. However, the amount of work performed per MT search differs, with MT searches with test values that are further from the values that are stored in the transposition table, incurring more work. So, looking back at the number of nodes visited in Figure 11, this does not mean that MTD-Bi actually performed less work.

[1]The measurements for MTD-F and MTD-Bi without conspiracy search were done on a version that improperly handled search instability, causing them to terminate too soon after an MT search, and thus perform too few MT searches. This had minimal impact on the number of nodes searched as can be seen in Figure 10. There is no difference in the number of MT searches performed between the non-conspiracy searches and their conspiracy search counterparts in the last 50 positions since search instability did not occur in those positions.

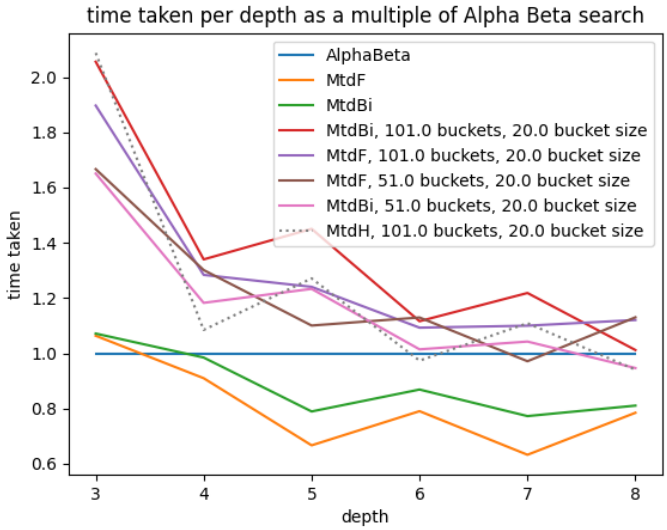

Figure 13: The time taken by each algorithm to complete calculating the positions on the last 50 positions.

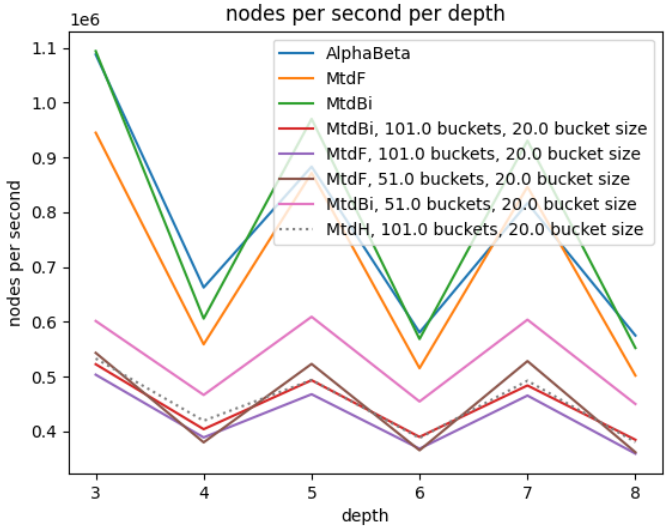

Figure 14: The nodes per second for each algorithm configuration tested.

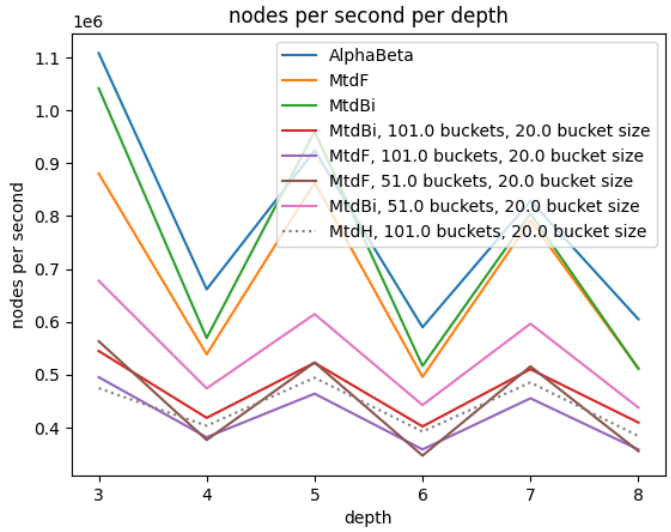

Figure 15: The time taken by each algorithm to complete calculating the positions on the last 50 positions.

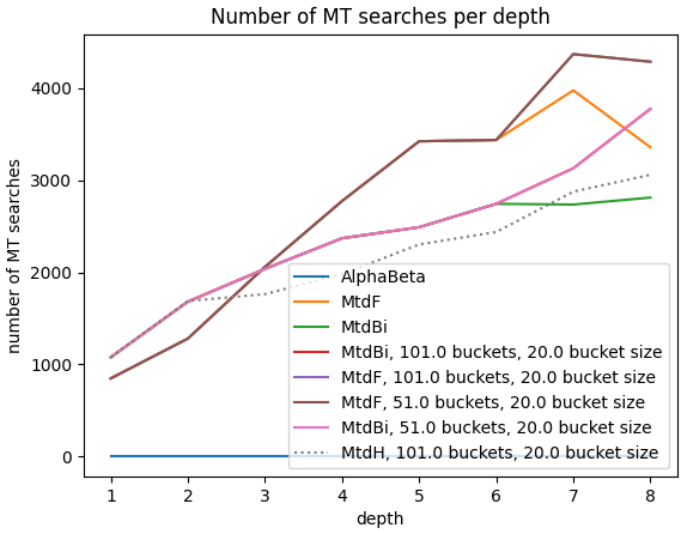

Figure 16: The number of MT searches performed for each algorithm configuration tested over the whole dataset.

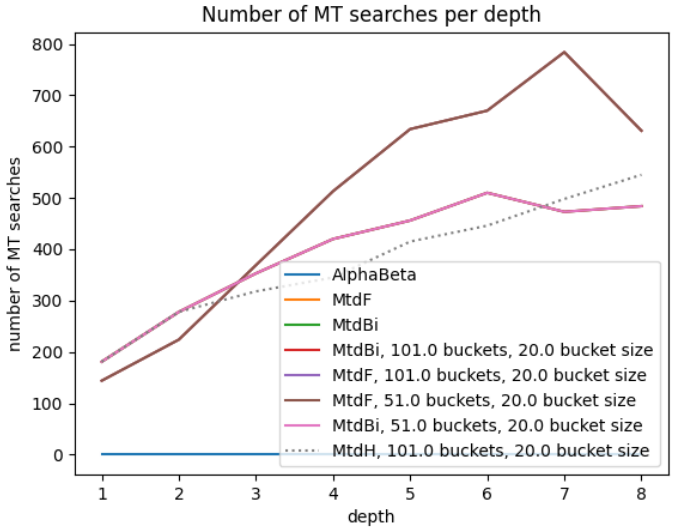

Figure 17: The total number of MT searches on the last 50 positions.

6 Discussion

One of the improvements that can be made to increase the efficacy of MTD-H would possibly be the use of quiescence search in position scoring. The advantage in number of nodes searched alternates between MTD-F and MTD-H on odd- and even depths. One explanation of the diminished predictive power of MTD-H compared to MTD-F at those odd depths, could be the way the best position is scored when quiescence search is absent: Since no further captures are considered, no recaptures are considered. This may lead to precarious situations, e.g. the best of line is found to be one where a queen is sacrificed for a knight at the last move, since no recaptures are considered.

Another possible improvement could be how the next test value is chosen based on the generated probability distribution. In the current implementation, the median of the generated probability distribution is chosen as the next test value, but this could be replaced by selecting the mean instead.

Furthermore, as can be seen in the difference between MTD-F and MTD-Bi and their conspiracy search counterparts, collecting the conspiracy numbers incurs a cost regarding the number of nodes per second being calculated. Under the current implementation, the conspiracy numbers are stored in two full arrays of buckets, one for the amount of conspirators needed to change the outcome of the game tree upwards, and for changing the outcome downwards. As the number of conspirators needed to change the outcome "upwards" towards a value that is lower than the current outcome of the game tree is guaranteed to be zero, those memory addresses are redundant. For changing the outcome "downwards" the same applies, but for the memory addresses that are represents buckets with a value that is higher than the outcome of the game tree. A smarter implementation of this would allow for halving the memory footprint needed for collecting the conspiracy numbers. One could suppose, that the performance impact of this, regarding nodes per second, would be comparable to changing the number of conspiracy buckets from 101 to 51, as shown in Figure 14. Furthermore, given the current implementation, one other possible for explanations to improve the nodes per second, could be optimizing the number of CPU cycles spent merging the collections of conspiracy numbers while traversing the game tree. Similarly one could look at reusing old, discarded allocations for conspiracy numbers in an arena, instead of continuously creating new memory allocations.

One last idea that could warrant more investigation would be storing more information in the Transposition Table: For example, one could consider storing conspiracy numbers, as well as estimated game tree size for

the positions at the first depth. This could allow for more sophisticated strategies balancing expected work of calculating the game trees for a given move, with the probability distribution of their eventual evaluation, sharing similarities with the search strategies such as Proof Number Search and its variants [4, 20, 22]. As an example, given that there exists a move with a small estimated game tree and a spike in probability for a high evaluation, one may consider evaluating that specific move only with a higher test value. This would also expand on work established by Plaat et al. [13] and McAllester [11].

7 Conclusion

MTD-H introduces conspiracy numbers to the MTD framework, and tests an implementation of this through the use of a chess engine. Although, due to cutoffs, during searching it cannot collect full information about conspiracy numbers, it uses the information generated by the conspiracy numbers to create a probability distribution of the final outcome of the game tree that is being searched. This information is then used to select the next test value for the MTD framework, in a more informed manner.

Overall, under the current implementation, MTD-H does not seem to outperform MTD-F. Although, this implementation of MTD-H is comparable regarding the number of nodes searched, the reduction in nodes per second, due to the collection of conspiracy numbers, makes it take longer with regard to computing time.

MTD-H uses a probability distribution to speed up the binary search used by MTD-Bi, and derives that probability distribution wholly from conspiracy numbers. MTD-H seems to require a lot less nodes searched than MTD-Bi to solve the same position, attesting to the usefulness of the information that conspiracy numbers convey to speed up searches from the MTD framework.

8 Appendix

A The Parameters for the MTD-H probability calculations

training_depth	target_depth	p	w_side_down	w_side_up	c
1	3	0.805	0.129	0.161	0.00831
2	3	0.985	0.010	0.129	0.00841
2	4	0.811	0.160	0.208	0.00764
3	4	0.994	0.071	0.082	0.00870
3	5	0.767	0.192	0.150	0.00791
4	5	0.999	0.067	0.111	0.00805
4	6	0.582	0.174	0.165	0.00807
5	6	0.999	0.051	0.051	0.00909
5	7	0.842	0.121	0.119	0.00882
6	7	0.999	0.041	0.116	0.00825
6	8	0.988	0.138	0.180	0.00778
7	8	0.999	0.027	0.067	0.00920

Table 1: The parameters used for the calculation of the probability distribution. Cut off at three digits. training_depth represents the depth from which the conspiracy numbers are used to predict for target_depth.

B Reproducing the MTD-H Results

```
Usage: store_analysis.exe [OPTIONS]

Options:
  -d, --db-path <DB_PATH>
          The path to of the DB to write to [default: sqlite://sqlite.db]
  -d, --dataset <DATASET>
          Which positions to analyze [default: win-at-chess] [possible values: win-at-chess, lichess-openings]
  -p, --play-options <PLAY_OPTIONS>
          Whether, for each position, to play a match or only that position [default: position] [possible values: match, position]
  -s, --search-depth <SEARCH_DEPTH>
          The depth to search to [default: 6]
  -a, --algorithm <ALGORITHM>
          Which search algorithm will be used [default: mtd-bi-iterative-deepening-conspiracy] [possible values: mtd-bi-iterative-deepening-conspiracy, mtdf-iterative-deepening-conspiracy, mtd-bi-iterative-deepening, mtdf-iterative-deepening, alpha-beta-iterative-deepening, mtdh-iterative-deepening]
  -b, --bucket-size <BUCKET_SIZE>
          Determines the width of the conspiracy buckets in Centipawns [default: 20]
  -n, --num-buckets <NUM_BUCKETS>
          How many buckets there are for conspiracy search. Must be uneven [default: 101]
  -m, --merge-fn <MERGE_FN>
          Which merge function to use for merging multiple mt_searches [default: merge-remove-overwritten] [possible values: merge-remove-overwritten]
      --neglect-transposition-table
          Disables the transposition table
      --minimum-transposition-depth <MINIMUM_TRANSPOSITION_DEPTH>
          The minimum entry depth required to be considered for the transposition table [default: 2]
      --mtd-h-params-path <MTD_H_PARAMS_PATH>
          The path for the mtd-h parameters [default: ./python/analysis_output/optimal_params.csv]
      --mtd-h-training-distance <MTD_H_TRAINING_DISTANCE>
          The default distance for the distance between training and target distance for MTDH params [default: 2]
  -h, --help
          Print help
  -V, --version
          Print version
```

Listing 6: The program options for running the executable to reproduce the results. Can be found at `https://github.com/nielsgroen/sn0l`

References

[1] Handling search inconsistencies in MTD(f). https://jhorssen.home.xs4all.nl/Maximus/research/Handling%20Search%20Inconsistencies%20in%20MTDf.pdf.

[2] Search instability. https://www.chessprogramming.org/Search_Instability, 2019.

[3] M. H. Albert, R. J. Nowakowski, and D. Wolfe. *Lessons in play: an introduction to combinatorial game theory*. CRC Press, 2019.

[4] L. V. Allis, M. van der Meulen, and H. J. Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[6] G. Chaslot, S. Bakkes, I. Szita, and P. 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.

[7] E. D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *International Symposium on Mathematical Foundations of Computer Science*, pages 18–33. Springer, 2001.

[8] J. C. Groeneveld. The Sn0l Engine. https://github.com/nielsgroen/sn0l, Aug. 2023.

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

[10] Lichess.org. Lichess Opening Database. https://github.com/lichess-org/chess-openings, Aug. 2023.

[11] D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988.

[12] J. Pearl. Asymptotic properties of minimax trees and game-searching. *Probabilistic and Causal Inference: The Works of Judea Pearl*, page 61, 1980.

[13] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Nearly optimal minimax tree search? *arXiv preprint arXiv:1404.1518*, 2014.

[14] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. A new paradigm for minimax search. *arXiv preprint arXiv:1404.1515*, 2014.

[15] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. SSS*= alpha-beta+ TT. *arXiv preprint arXiv:1404.1517*, 2014.

[16] F. Reinfeld. *Win at Chess:(formerly Titled Chess Quiz).* Dover Publications, 1958.

[17] A. Saffidine and T. Cazenave. Multiple-outcome proof number search. In *ECAI 2012*, pages 708–713. IOS Press, 2012.

[18] M. Seo, H. Iida, and J. W. Uiterwijk. The PN-search algorithm: Application to tsume-shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.

[19] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[20] H. J. Van Den Herik and M. H. Winands. Proof-number search and its variants. *Oppositional Concepts in Computational Intelligence*, pages 91–118, 2008.

[21] J.-J. Van Horssen. Move selection in MTD (f). *ICGA Journal*, 41(1):15–23, 2019.

[22] M. H. Winands, J. W. Uiterwijk, and J. van den Herik. PDS-PN: A new proof-number search algorithm: Application to lines of action. In *Computers and Games: Third International Conference, CG 2002, Edmonton, Canada, July 25-27, 2002. Revised Papers 3*, pages 61–74. Springer, 2003.

[23] A. L. Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13(2):69–73, 1990.