A strong Clobber 2D Player using CGT MCTS-Solver

Abdul Wahab (wahab1), Muhammad Kamran Janjua (mjanjua), Saqib Ameen (saqib1)

Team Name: TryCatch

1 Introduction

In this project, our goal is to couple combinatorial game theory (CGT) enhancements such as endgame database, and subgame simplifications with the Monte-Carlo Tree Search (MCTS) Solver to design a strong 2D-Clobber player.

Clobber is a two-player game invented by Albert, Grossman, Nowakowski and Wolfe (Albert et al., 2005) in 2001. It is played on a rectangular grid containing alternating stones of black and white color to form a checkered pattern in its initial state. Each player owns stones of one color representing the player. Players take alternating turns to make a move. In a move, a player replaces opponent's stone placed in the adjacent position. Last player who is able to make a move wins the game. For each move of a player, its opponent also has a move. Hence, unlike other games, such as domineering or amazons, there is no concept of being ahead or behind on the basis of number of moves. As a result, in CGT, Clobber is classified as all-small games (Albert et al., 2019), i.e., all positions are infinitesimals.
The game of Clobber can either be one dimensional (1D) or two dimensional (2D) where the dimension is given by $n \times m$ where n represents the number of rows and m represents the number of columns. The game has been studied in the literature to the extent that there is also a Clobber tournament where computer program based players compete with each other (Willemson and Winands, 2005).
In this project, we extend the previous work done by (De Asis, 2017) on designing a Monte-Carlo Tree Search (MCTS) player using CGT in the following ways:

1. We design a 2D-Clobber player using MCTS Solver (Winands et al., 2008) and CGT techniques. Instead of computing values for infinitesimals, we use endgame database to simplify the games.
2. We extend the database design of (Hernandez, 2017) to have an endgame database of the infinitesimals using sub-zero thermography. Previously, only symmetry and inverses were considered. We consider 16 different variations of the board (discussed in Section 4).

Lastly, we explore a side idea where we plug-in Artificial Neural Network (ANN) within the MCTS during rollouts when the NN thinks a board position is a win (Cotarelo et al., 2021). For evaluation, we evaluate our work by comparing it with MCTS-CGT player of (De Asis, 2017). We also conduct a few case studies to analyze how player's performance changes with different enhancements. We hope this work will help future researchers to design a better Clobber player.

2 Literature Review

The game of Clobber has been studied both under the aspects of designing a solver and a player for the game. The work done by Wieczorek et al. (2011) designed four heuristics based on the number of stones of player's color in sub-games, looking at different move ordering for patterns, etc. These heuristics were designed to be used in the early and the middle phase of the game. In the work authors conclude that these heuristics failed often due to their precise design nature, and that there is little information regarding good play. Griebel and Uiterwijk (2016) improved a NegaScout solver by coupling it with an endgame database to look up the values of the stored positions instead of solving them. However, the work done had little impact with regards to playing on Clobber positions that cannot be looked up in a database and lacked CGT enhancements. L. and A. (2017) worked on designing a NN based Clobber player. However, they concluded that NN based players have yet to defeat a vanilla MCTS player. A recent work done by Cotarelo et al. (2021) looks at using a neural network during the rollout phase for the game of Dots-and-Boxes as an example. If the result is win with a probability greater than a threshold value, then a win is backpropagated, otherwise random simulations are performed to get the value estimates. We tried to implement a similar idea where we try to use NN to skip simulations based on the inference given by the NN. However, this did not work out as expected — details are discussed later in the report.

Given the efficacy of employing MCTS-based player, the Clobber playing program, MC-Mila, presented in the Clobber tournament (Willemson and Winands, 2005) achieved great success. Instead of standard MCTS, the Clobber player MC-Mila uses MCTS-Solver (Claessen, 2011, Winands et al., 2008). To avoid too many simulations, MCTS-solver is able to solve the game-theoretic value of a position and incorporate this value in the search tree, so that it can be used for future simulations.

In a course project, De Asis (2017) designed a CGT-aware MCTS based Clobber player. The player uses CGT techniques for subgame simplifications prior to the MCTS simulations. Simplification step includes zero game removal, and replacement of complex games with equivalent simple games. An alpha-beta solver was used to compute the infinitesimals on-the-go instead of looking up the values in an endgame database. Apart from this, during our work, we found a few bugs and limitations in this previous work.

The backpropagation implemented in the MCTS was based on the one-skip backpropagation wherein during backpropagation only those nodes which belonged to the node's player were updated. In other words, the parent of the current node was skipped during backpropagation since it was its opponent's node and its value was not updated, only the node visit count was updated. We found that one-skip backpropagation hindered the performance of MCTS.

Additionally, we found that some cases were not handled properly:

1. Sometimes during the simplification, all the allotted time per move was consumed. As a result, MCTS did not have anytime to select a node and it was returning an illegal move, `a1a1`, all the time. We fixed that to return the first child.

2. No check was placed on illegal moves in the Clobber Framework. For instance, for some reason, it was playing `a1a2` move multiple times. While we could not find the exact reason for this problem, we placed a check to identify such moves and stop the game. Further, since we wrote most of MCTS-Solver (our player) from scratch, we found that to the best of our knowledge, this problem does not exists in our player.

3 Clobber Framework

We use the Clobber framework written by De Asis (2017) and Hernandez (2017) in C++. It uses a bit-board representation for the board where each player is denoted by an integer — 0 for Black, 1 for White. The bit-board representation allows efficient manipulations. This representation can also be exported to string where 'x' represents black, and 'o' represents white and '.' indicates empty spaces. The string representation of the board follows the row-by-row iteration of the board. Consider a 2×2 Clobber board which can be represented in string as: r2c2xoxo, where r2 and c2 indicate 2 rows and 2 columns respectively.

The initial move list is generated by iterating through every board index and looking at the stone colors. After a move is played, the available move list is updated by looking at the the move taken and by performing a pass through the list to identify which ones are no longer available and which new moves should be added. The resultant moves are flipped for the opposing player. Furthermore, the sub-games are extracted by computing the bit masks of connected stones and applying those masks to the main board. For more details, the report of De Asis (2017) can be consulted.

4 Database Design

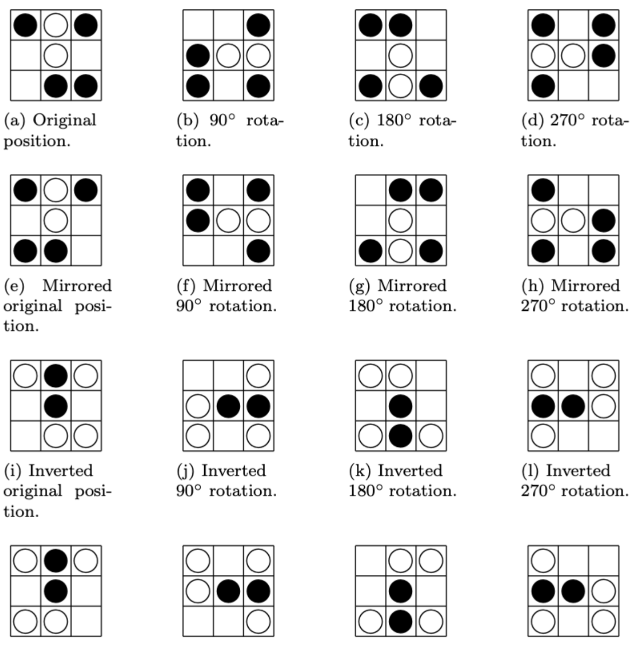
The technique for database design is adopted from the work of Hernandez (2017), however we extended it to include more variations of the boards. The design is briefly discussed here.
The endgame database is computed using subzero thermography. For each game, a flag, the exact value, lower bound, upper bound, and its board along with the dimensions is saved. A sample board entry looks like this: 1 0 0 0 1 0 0 0 0 0 0 0 0 r1c2xo. The first entry (flag), represents whether or not we know the exact value. The next four entries represent the exact value of board as an integer multiple of $\uparrow$, $\uparrow^2$, $\uparrow^3$, and $\star$. For instance, the above mentioned entry of 0 0 0 1 shows that it is a star game. Note that the integer multiple can be negative as well. This allows us to represent both sides of the comparison scale in single entry. Similarly, the next four entries are for lower bound, and the next four are for upper bound. In the end, we have board representation where 'r' and 'c' followed by a number represent row and column numbers, 'x' represents black, and 'o' represents white. Empty spaces are represented by '.'.
To work out the values of the game, the game G is compared to several different reference games h_k, for which value and their thermographs are known. The set of reference games $H = h_k$ is selected such that the inverse of each element of H exists, the elements of H are not confused with each other, and a partial order exists between them. To find a value, thermograph of a game G and reference games h_k are added. Depending on the outcome, 0, > 0, < 0, and fuzzy, we know if the game is inverse, greater, less, or confused respectively. First we use the following scale for comparison starting from zero.

$$\Uparrow \;>\; \uparrow + \uparrow^2 \;>\; \uparrow + \uparrow^3 \;>\; \uparrow \;>\; \uparrow^2 \;>\; \uparrow^3 \;>\; 0 \;>\; \downarrow^3 \;>\; \downarrow^2 \;>\; \downarrow \;>\; \downarrow + \downarrow^3 \;>\; \downarrow + \downarrow^2 \;>\; \Downarrow$$

For the sake of simplicity, we limit the scale to exponent $k = 3$. If G is confused on this scale, a second scale based on $\star$. This way, we either end up computing an exact value, or a better upper/lower bound. For $\star$, the fuzzy area lies between $\downarrow$ and $\uparrow$. That means that the value of $\star$ is between $\downarrow$ and $\uparrow$, but the exact value is not known. A value $n \uparrow + \star$ is positive if $n > 1$, negative if $n < -1$ and fuzzy if $n = -1$, 0 or 1 (e.g. $\uparrow\star$ is fuzzy, but $\Uparrow\star$ is positive) Claessen (2011). The second scale used is shown below:

$$\Uparrow\star \;>\; \uparrow\star \;>\; \star \;>\; \downarrow\star \;>\; \Downarrow\star$$

This scale is limited to the values shown above. Using these two scales, combined with the rules described in Section 3.3 of Hernandez (2017), we compute exact, upper bound, lower bound values. The database contains the values of boards with at most 8 connected stones. This limit of 8 is set due to the most frequent occurrence of infinitesimals in such boards, as identified in previous studies by Hernandez (2017) and De Asis (2017).
The entry from the database is retrieved and an evaluation function returns either 0 (for left player) or 1 (for right player). If the retrieved entry's value is exact, then the game is evaluated according to this exact value. If the exact value is unknown, then depending upon whose turn is it (left/right player), the bounds are evaluated. If it is *right player*'s turn and exact value is unknown, and upper boundary is ≤ 0 or $\star$, then result is 1. Otherwise, if lower boundary is ≥ 0 or 0, the result is 0. The reverse holds for the left player's turn.
It is observed that the value of a board, and the value of its 90° rotation, 180° rotation, 270° rotation, and their mirrors is same. Similarly, the value of the inverse of board, and the inverse of its 90° rotation, 180° rotation, 270° rotations and their mirrors, is same but opposite to the original board. This means that by storing a value of a single board, we can fetch the values of its all 16 different variations. However, we noted that computing all 16 variations on the go was time consuming since we use it inside our MCTS simulations. So, instead of computing all the 16 variations on-the-go to determine the value of the board, we store all the values in the database. The figure 1 shows one such example for a 3×3 board position.


(a) Original position. (b) 90° rotation. (c) 180° rotation. (d) 270° rotation.

(e) Mirrored original position. (f) Mirrored 90° rotation. (g) Mirrored 180° rotation. (h) Mirrored 270° rotation.

(i) Inverted original position. (j) Inverted 90° rotation. (k) Inverted 180° rotation. (l) Inverted 270° rotation.

(m) Inverted and mirrored original position. (n) Inverted and mirrored 90° rotation. (o) Inverted and mirrored 180° rotation. (p) Inverted and mirrored 270° rotation.

Figure 1: All board variations of a 3×3 board position. (Claessen, 2011)

5 CGT Enhancements

Before calling the MCTS, we simplify the (sub)games using CGT techniques. When it is our turn to play, we identify all the sub-games, and query their infinitesimal values from the database. Based on the values, we remove the sub-games which add up to zero, or which are zero, because they do not effect the result of the game. For instance, if one sub-game is $\uparrow$, and other sub-game is $\downarrow$, then we can remove both, since $\uparrow + \downarrow = 0$. Similarly, even number of $\star$ games are removed, and odd number of $\star$ games are replaced by a single $\star$ game. In the end, this simplified board is fed to MCTS-Solver to further evaluate and select a move.

6 Monte-Carlo Tree Search (MCTS)

Starting from a given position, MCTS randomly grows tree in four steps: selection, expansion, playouts/simulations, and backpropagation. In selection step, it recursively traverses the tree by selecting nodes based on a selection criteria until the end of the tree. We use UCT function for selection. Once a node is selected, it is expanded and all children are added to the tree. After that a child is randomly selected and simulation is done. In simulation, game is played randomly and its value is determined. That value is then backpropagated to update the values of all the nodes visited from root till that node. During the updates the number of visits of nodes are also updated. These steps are repeated a fixed number of times, or until it runs out of time. After that, a move is selected using the formula $\frac{v}{n}$, where v is the value of the node, and n is the number of visits. This is how the vanilla MCTS-player works. These steps are also illustrated in the Figure 2. De Asis (2017) implemented the CGT enhancement as discussed in Section 5 using an alpha-beta solver on-the-go, however we replaced the solver with the endgame database. We further extended it to implement an MCTS-Solver as discussed in the following section.

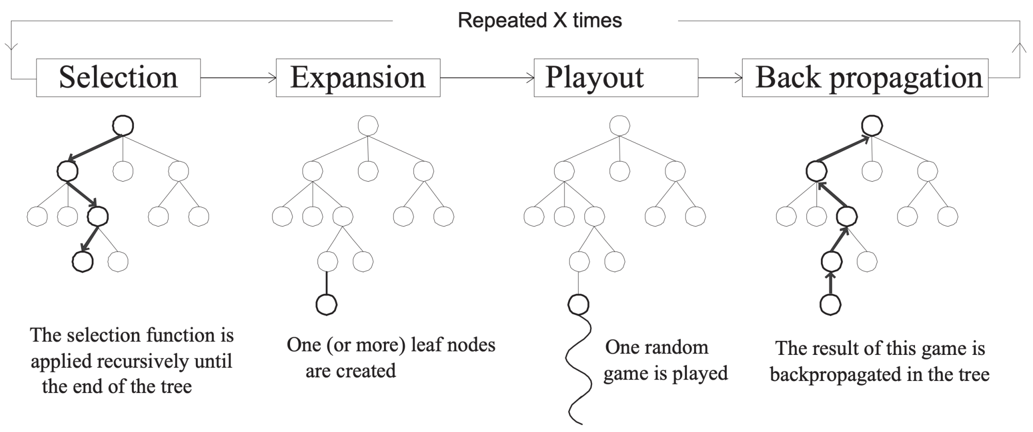

Repeated X times
Selection
Expansion
Playout
Back propagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One random game is played
The result of this game is backpropagated in the tree

Figure 2: Overview of the steps of MCTS (Claessen, 2011)

6.1 MCTS-Solver

MCTS-Solver originally proposed by Winands et al. (2008) for sudden-death games like Lines of Action (LOA), is a powerful technique that combines the power of a game solver with MCTS. MCTS-solver uses the game-theoretic values of nodes to improve the selection (of MCTS), and the final-move selection. To implement the MCTS-Solver, the backpropagation needs to be modified to cater for positive and negative infinities in addition to the value estimates of each node. The positive and negative infinities correspond to proven wins and losses respectively. The infinities are backpropagated in a negamax fashion, i.e., a node is a proven win if one of its children is a proven win, and a node is a proven loss if all of its children are a proven loss. Also, a proven win for a player is a proven loss for the opponent.

Backpropagation. In our implementation of the MCTS-Solver, we update and backpropagate value estimates and infinities for a node with respect to its parent. For example, if a node has a value of ∞, then that node corresponds to a winning action for its parent. Similarly, if a node has a value of $-\infty$, then that node corresponds to a losing action for its parent. In our implementation we do not set the value of a node to ∞, rather we set a flag denoting either a ∞ or a $-\infty$. We backpropagate infinities in a negamax fashion. If a node is a proven win (∞) then the parent of that node would be a proven loss (-∞). This is because the parent of the node is a proven loss with respect to the parent of the parent (grandparent of original node). However, in case if a node is a proven loss ($-\infty$), the parent will get a ∞ (i.e., proven win for the grandparent of original node) only if all the siblings of the node are proven losses (-∞), otherwise regular win is backpropagated. The case of propagation of infinities is shown in the 3.

Selection. The advantage of identifying proven wins and losses, and setting the corresponding infinity flags of the nodes is that during selection we can now make better selection choices, i.e., we ignore the nodes which are proven losses, and incentivize nodes which are proven wins. We are setting the infinity flags regardless of which player it is to play in the selected leaf node. This allows our player to simulate the best behavior even for the opponent and return the corresponding best response. This enhancement reduces the number of nodes that are explored in the selection phase, and improves the path taken during selection.

Final Move Selection. For final move selection after the MCTS simulations, we use the same approach as used in Winands et al. (2008), called *secure child*. Instead of the regular selection criteria ($\frac{v}{n}$), a slightly different move selection criteria is used:

$$v + \frac{A}{\sqrt{n}}$$

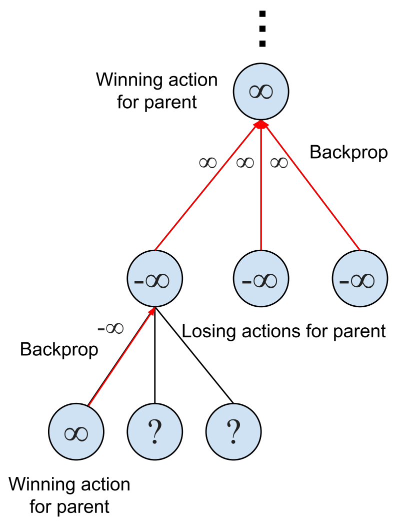

Winning action for parent
∞
∞
∞
∞
Backprop
-∞
-∞
-∞
-∞
Backprop
Losing actions for parent
∞
?
?
Winning action for parent

Figure 3: Propagation of infinities for proven nodes in MCTS-Solver. In the lower sub-tree, one of the leaf nodes is a ∞, so we backpropagate a $-\infty$ to its parent. In the upper sub-tree, in order to backpropagate a ∞ we check if all its siblings have $-\infty$ only then ∞ is backpropagated to the parent.

Here, v is the node's value, n is its visit count, and A is a constant. Based on the work of Winands et al. (2008) A is fixed to 1 here.

Skip Simulations. Another enhancement that we do is that once a leaf node gets selected, we check if that node is a proven node, then we skip the random simulations for getting value estimates, rather we simply backpropagate the value. This saves time by not doing random playout and allows for more MCTS simulations to happen.

Early Stopping. Another core enhancement that we do in our implementation of the MCTS-Solver is the early stopping of MCTS simulations. This is done when a selected leaf node is a proven win and is also a child node of the root node (denoting the current game board). In such a situation we simply pick the action corresponding to the proven win child node, and do not need to do further simulations.

Identifying proven nodes. Our clobber player identifies proven nodes by using the CGT techniques and the CGT-based database explained in Section 4, for each selected node within the MCTS simulations. Given a selected node, we identify the subgames, and then query the database to get the infinitesimals of all the subgames, we then evaluate these infinitesimals to identify the value of the game, i.e., 0 (Black wins), 1 (White wins), -1 (Not known). To avoid making unnecessary queries to the database we check if the size of the biggest subgame is less than or equal to the biggest board stored in the database. Once the value of the node is identified, we look at the player of the parent of that node and set the ∞ or $-\infty$ according to that player. For example, if the value of a node is 0 (Black wins), and the player of the parent of that node is also Black, then we set the value of that node to ∞. This means that the Black player at the parent node can lead to a proven win by taking the action corresponding to the proven node.

7 Experiments & Results

In this section, we discuss the experiments conducted and analyze their results. All of the experiments and studies are performed against CGT-MCTS player (De Asis, 2017). CGT-MCTS-Solver refers to our final MCTS-Solver with all the CGT enhancements and endgame database.

Board Size →	6×6	8×8	10×10
ZeroDB	49	58	55
BProp	74	93	85
BProp + SC	81	90	92
CGT-MCTS-Solver	83	87	88

Table 1: Win ratio (%) for 100 plays with 30s move time, on different board sizes with different enhancements all played against CGT-MCTS: ZeroDB — subgames simplification uses endgame database, BProp — improved backpropagation in MCTS, BProp + SC — BProp with secure child method for final move selection, CGT-MCTS-Solver — MCTS-Solver player using all CGT enhancement and endgame database.

Time (s) ↓	6×6	8×8	10×10
10	85	90	91
15	84	87	93
30	83	87	87

Table 2: Win ratio (%) of CGT-MCTS-Solver when played against CGT-MCTS Player for 100 plays under different times.

Analysis of Different Enhancements. The first set of experiments consists of comparing the win ratio (%) of all the different enhancements with the CGT-MCTS player. The results are listed in the Table 1. The first enhancement is coupling the ZeroDB, where we query the database before the MCTS step to simplify the board by removing zero games. In case of large board sizes, such as 8×8 and 10×10 the win rate of our player improves as compared to the base line. Since on larger board sizes, bigger sub-games are formed hence, we are able to quickly query the values from database to simplify the boards. However, in case of 6×6 board, the win ratio is comparable because on smaller boards, smaller sub-games are formed, and both the alpha-beta solver of CGT-MCTS and CGT-MCTS-Solver do equally well on simplifying such games.
Next we discuss the improved backpropagation (BProp), which fixes the backpropagation issue in CGT-MCTS and performs better on all board sizes with comparatively larger gains on the large board sizes. This improved backpropagation is combined with the secure child (SC) method of selecting the final move which further improves the win rate for all board sizes. This shows that secure child is a better final move selection function as compared to the regular move selection criteria. Another enhancement is the implementation of CGT-MCTS-Solver. It combines all the previous enhancements and implements MCTS-Solver as discussed in the Section 6.1. We perform consistently better on all board sizes with win ratio increasing from smaller board size to larger board size i.e., 83% on 6×6, 87% on 8×8, and 88% 10×10. With MCTS-Solver, we achieve the highest win ratio on 6×6 board as compared to other enhancements. We believe this is attributed to the fact that on smaller board size (6×6), MCTS-Solver is able to find the proven wins/losses easily and is able to better compete with the CGT-MCTS player. The win ratio increases on the larger board sizes and is comparable to the other enhancements, if not better. This is because we have to continuously query the database and evaluate the sub-games to identify a proven loss or win within the MCTS. Further, Table 2 quantifies how our player performs against CGT-MCTS on different move times. Results show that it performs equally well on different move times. We believe that this is mainly because we avoid redundant simulations in MCTS-Solver and do early-stopping when we encounter proven nodes. While the process of finding proven nodes is slow, once found, they can significantly help speed up the process and thus the MCTS-Solver needs overall less time. We also perform an extended study to verify this hypothesis.

Comparison of Simulations and Nodes Explored. The role of MCTS-Solver and our enhancements become more apparent and significant when we analyze the number of MCTS simulations and the number of nodes explored during selection phase of MCTS. Owing to the MCTS-Solver, Skip Simulations, and Early Stopping enhancements, see Section 6.1, our player does less MCTS simulations and visits less nodes in selection phase of MCTS to find the suitable path and

Avg. # of ↓	CGT-MCTS-Solver	CGT-MCTS	Relative Decrease (%)
Simulations	15283611	105058096	85.45
Nodes explored	1827574613	5701559775	67.94

Table 3: Comparison of number of average number of simulations performed and nodes explored per game and relative decrease between CGT-MCTS-Solver and CGT-MCTS player.

a leaf node. This can be seen in the Figure 4 and Figure 5, that the CGT-MCTS-Solver player consistently visits less number of nodes in the selection step of MCTS, and does less number of simulations to return the optimal action. Table 3 shows the average (over 100 games) number of simulations and nodes visited of our CGT-MCTS-Solver compared with the CGT-MCTS, and the relative percentage decrease. This analysis shows that our CGT-MCTS-Solver player requires less time to return the optimal action. It is important to note that we specifically design our player to stop early for the sake of this analysis i.e., whether our player requires less simulations and visits less nodes. However, we believe that doing more simulations even after the early stopping criteria is met should not effect our results.

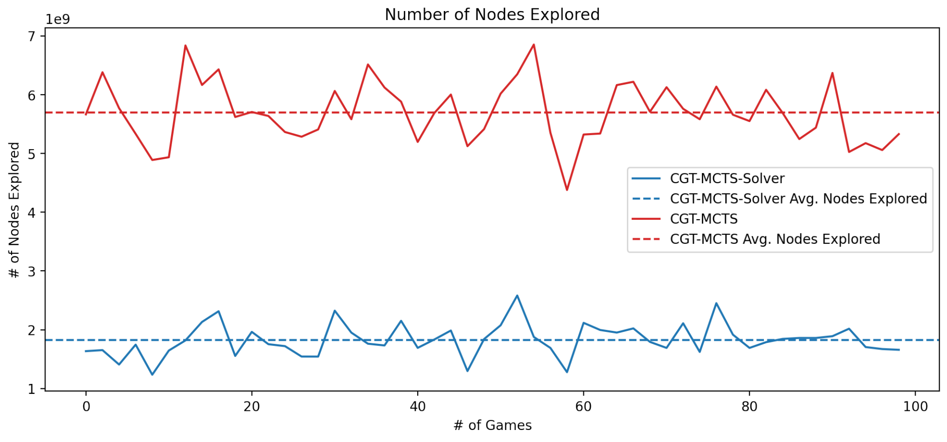


Figure 4: Comparison of average number of nodes explored in the selection phase of MCTS.

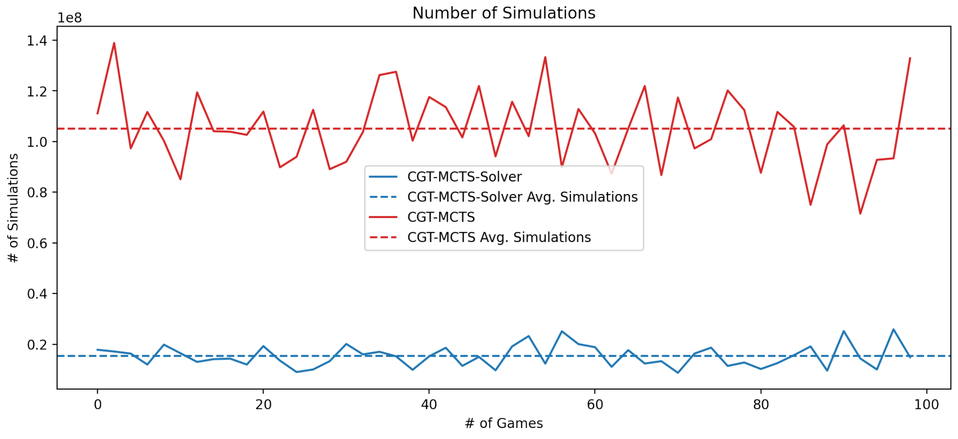


Figure 5: Comparison of average number of MCTS simulations performed.

Comparison of CGT-MCTS-Solver and CGT-MCTS Player on Fixed Times We verify the claim above by conducting an experiment where we allow less time for our CGT-MCTS-Solver and more time to the CGT-MCTS player. In Figure 6, on the left hand-side, when we allow 30 seconds for the CGT-MCTS player, and vary the move time for our CGT-MCTS-Solver $(30, 15, 10)$, we observe that our player consistently performs better even when given less time. The difference in the win-ratios is negligible and can be due to the limited number of runs (100). On the right hand-side of the Figure 6, when we allow 15 seconds to the CGT-MCTS player and vary the move time for our CGT-MCTS-Solver $(30, 15, 10)$, we see that there is a decrease in the win-ratio (although much higher than the CGT-MCTS), as we reduce the move time for the CGT-MCTS-Solver player. This empirically shows that our CGT-MCTS-Solver player requires less time to play well compared to the CGT-MCTS player. One interesting thing we observe during this experiment is that the CGT-MCTS player performs better when it is given 15 seconds for a move, compared to when it is given 30 seconds. This observation is consistent with what De Asis (2017) observes, i.e., CGT-MCTS player performs better when given 15 seconds. The author claims that this is due to the randomness or noise in the value estimates when given less time.

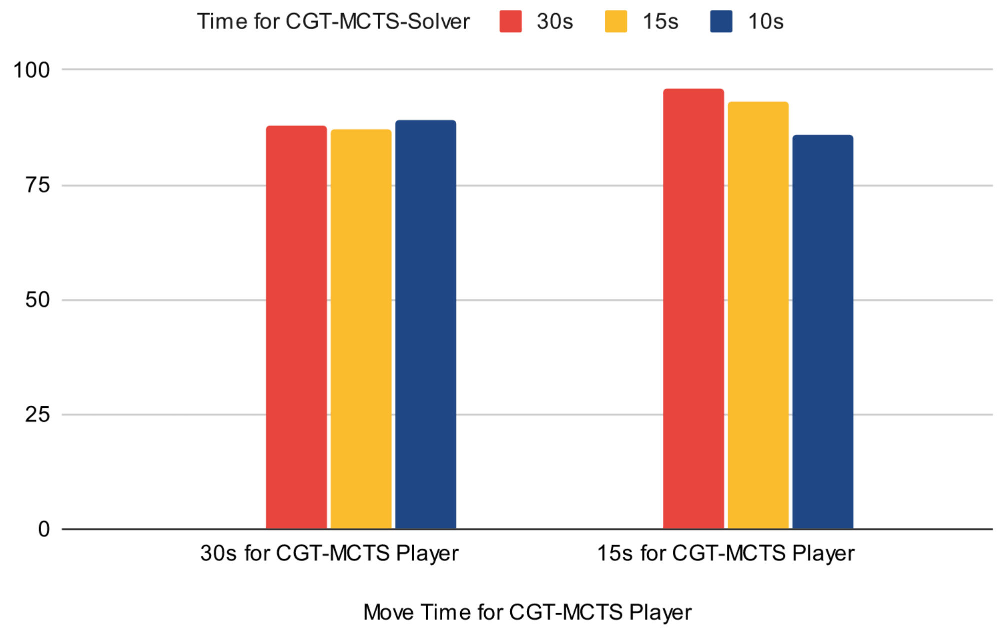


Figure 6: Win ratio analysis: CGT-MCTS player given fixed move times (30s and 15s) as the move times of our CGT-MCTS-Solver player varied (30s, 15s, and 5s) across 100 games on 10×10 board size.

8 Extended Summary: Neural Network within MCTS

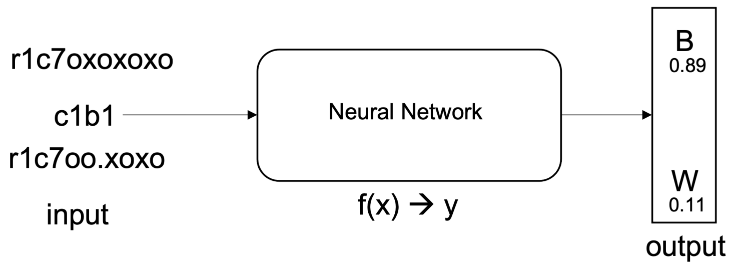


Figure 7: The artificial neural network design. The input consists of initial board configuration, the winning move, the board after the move has been made on it and the output is the outcome class. The input is x and the output is y, whereas the neural network serves as a function approximator $f(x)$.

A recent work done by Cotarelo et al. (2021) looks at optimizing the simulation phase of MCTS by coupling a neural network using the game of Dots-and-Boxes as an example. Following the

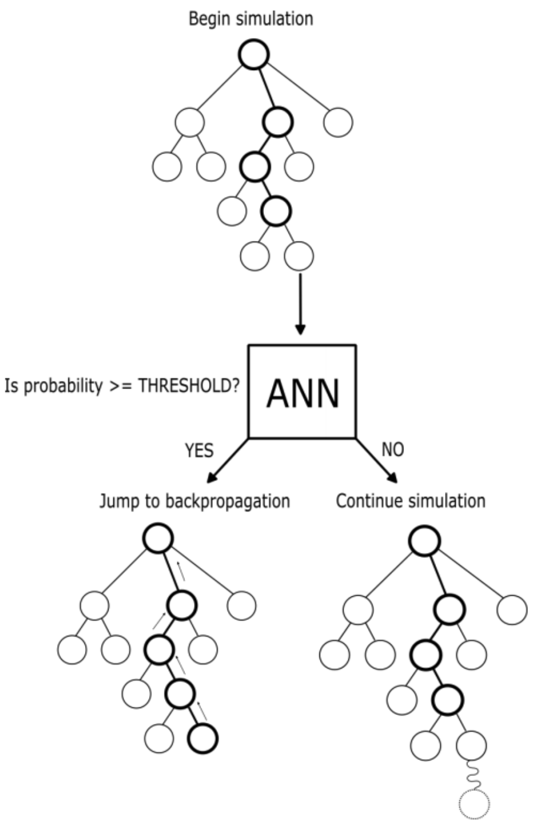


Figure 8: The flow-chart of coupling ANN within the MCTS. In the simulation step, selected node for simulations is fed to the ANN. If it is a win for our player with probability greater than a set threshold we go to backpropagation, otherwise perform simulation like normal setting. The figure is re-produced here from Cotarelo et al. (2021).

optimization suggestions of Fu (2016), the NN is used at the beginning of the simulation phase to predict the winner in the work of Cotarelo et al. (2021). If the result is consistent enough i.e. greater than some threshold value, that value is used and backpropagation phase is called, otherwise random simulations are performed. To prepare the data, we let run an alpha-beta solver for Clobber games of board sizes: $\{1 \times 1, 1 \times 2, .., 1 \times 10\}$, $\{2 \times 1, 2 \times 2, ..., 2 \times 10\}$, $\{3 \times 1, .., 3 \times 7\}$, and $\{4 \times 1, .., 4 \times 5\}$. Solving for boards on sizes larger than 4×5 was time consuming therefore we did not consider those. The dataset consists of the initial board configuration, the move, the board after the move has been made on it and the outcome class (win for black or white). We design a 2-layer artificial neural network (ANN) with softmax applied to last layer that takes the initial board configuration, the move, and the resultant board as input and learns a map between the outcome. The visual representation of the design is shown in Figure 7.
Following the implementation suggestions in Cotarelo et al. (2021), we couple the neural network before the simulation step. Once we are at the simulation step, we query the ANN before the simulation by fetching the input information from the node, passing it to the ANN and computing the outcome and the probability. We check if the outcome is in our favor and the probability of the output is greater than a certain fixed threshold (we fixed it to 0.5), then the backpropagation step is called; otherwise the simulations occur like normal. A figure adapted from the work in Cotarelo et al. (2021) is re-produced here for visual purposes, see Figure 8.
We evaluate the performance of NN-MCTS player against CGT-MCTS on different board sizes ranging from 3×3, 3×5, to 4×4 for a total of 100 games for each board size. We only query on flattened board sizes less than 17 due to the input format that the trained ANN accepts. If the player plays on the 3×5 board size, then the ANN can be queried right from the start since when flattened the length is less than 17.

3x5		4x3		4x4	
NN-MCTS	CGT-MCTS	NN-MCTS	CGT-MCTS	NN-MCTS	CGT-MCTS Player
37	63	57	43	19	81

Table 4: Results of NN+MCTS player compared to CGT-MCTS on three different board sizes for 5 second move time.

On experimentation, we found that the results were not significant with the current ANN design and the coupling with MCTS mechanism. We conducted experiments on board sizes 3×5, 4×3, and 4×4 with 5 seconds move time, see Table 4. We see that the results are worse compared to CGT-MCTS player. We hypothesize that this is because the current ANN design does not optimally utilize the position information and can be better designed. Another thing to consider is the time taken in ANN inference. Since the ANN requires the data to be presented in a certain format, we encode the string representations of boards and moves and pass it to the ANN which then takes some time to run the inference and return the result. This added computational overhead also limits the performance since in the time this entire process takes place, MCTS is able to run several simulations to get to better value estimates of the node. We did not evaluate the performance on larger board sizes and different move times since we did not observe encouraging initial results. Therefore, we conclude that coupling of ANNs with MCTS requires further investigation and careful design choice considerations before conclusive discussions can be made.

9 Limitations

To the best of our knowledge, results presented in this report are accurate and are reproducible from the given code base. However, a good amount of time was spent in debugging, fixing the issues, and verifying them in the previous code. As we developed our enhancements, many times we found that they did not work as expected. As a result, we had to rigorously debug and check for issues we encountered, as detailed in Section 2. For future work, we believe it would be better to have the framework in a simpler programming language like Python, for easy understanding, and prototyping in addition to having it fully documented.

10 Future Work

In this work we improved CGT enhancements using a database for simplifying the game board. In addition we combined CGT techniques within the MCTS part to improve the selection phase of MCTS and time efficiency of the player. However, the database we use can be further improved by adding more board positions. Improving the database would also further improve the MCTS-Solver which relies on the database to identify proven nodes.
Also, in this work we compared our CGT-MCTS-Solver mainly against the CGT-MCTS player, as a future work the proposed player can be tested and compared against other strong Clobber players. Another future work direction concerns with the ANN within MCTS, as detailed in Section 8. We believe that improving the ANN design to adapt to larger board sizes with efficient inference would help with win ratio, nodes explored and player speed. One possibility is to learn an embedding layer on the board representations to build latent board representations which can then be fed to an ANN to learn the map between this input and the outcome. Furthermore, exploring different move times and analyzing their effects in NN-MCTS player can also be investigated further.

11 Conclusion

In this project, we developed a 2D-Clobber player using MCTS-Solver and CGT-techniques i.e., an endgame database based on infinitesimals to perform subgame simplifications instead of using a solver on-the-go. In addition, we used CGT-techniques in the MCTS-Solver to identify the proven nodes using the database. We improve the player's performance in terms of increased win-ratio and reduction in the total number of nodes explored and number of MCTS simulations done as compared to the CGT-MCTS player De Asis (2017). We extensively evaluated our player on different move times, and board sizes. We ablated and detailed each modification and improvement made to our player to understand its effects on the evaluation metrics. We concluded that our MCTS-Solver player was able to consistently win against the CGT-MCTS player across all board sizes and move times. Lastly, we also explored a side idea where we implemented an ANN within MCTS before the simulation phase and evaluated its performance.

12 Authorship Acknowledgment

We acknowledge the work done by De Asis (2017) and Hernandez (2017) on Clobber player and solver respectively. Their code base provided us a base to do our work. We also acknowledge Prof. Martin for providing us the resources and helping us navigate through the project.

References

Michael H Albert, JP Grossman, Richard J Nowakowski, and David Wolfe. An introduction to clobber. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 5(2), 2005.

Michael H Albert, Richard J Nowakowski, and David Wolfe. *Lessons in play: an introduction to combinatorial game theory.* CRC Press, 2019.

Jeroen Claessen. Combinatorial game theory in clobber. *Masters-thesis Maastricht University*, 2011.

Alba Cotarelo, Vicente García-Díaz, Edward Rolando Núñez-Valdez, Cristian González García, Alberto Gómez, and Jerry Chun-Wei Lin. Improving monte carlo tree search with artificial neural networks without heuristics. *Applied Sciences*, 11(5):2056, 2021.

Kristopher De Asis. A cgt-informed clobber player. *CMPUT-655 Project Report, University of Alberta*, 2017.

Michael C Fu. Alphago and monte carlo tree search: the simulation optimization perspective. In *2016 Winter Simulation Conference (WSC)*, pages 659–670. IEEE, 2016.

Janis Griebel and JWHM Uiterwijk. Combining combinatorial game theory with an α-β solver for clobber. In *BNAIC*, pages 48–55, 2016.

J. Fernando Hernandez. Using left and right stop information for solving clobber. *CMPUT-655 Project Report University of Alberta*, 2017.

Damhuis L. and Kosters W. A. Convolutional neural network for clobber. *Leide Institute of Advanced Computer Science Posters Bachelorkla*, 2017.

Wojciech Wieczorek, Rafał Skinderowicz, Jan Kozak, and Przemysław Juszczuk. New trends in clobber programming. *ICGA Journal*, 34(3):150–158, 2011.

Jan Willemson and Mark Winands. Mila wins clobber tournament. *ICGA Journal*, 28(3):188–190, 2005.

Mark Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. pages 25–36, 09 2008. ISBN 978-3-540-87607-6. doi: 10.1007/978-3-540-87608-3_3.