Universiteit Leiden

Analyzing Strategic Methods for First-Player Wins in Chomp

Anass el Guermat (s3346668)

Supervisors:
Rudy van Vliet & Mark van den Bergh

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 4, 2026

Abstract

In this thesis, the game of Chomp is investigated mathematically and computationally. We go through various well-known strategies, expand them, and provide more novel generalizations of winning strategies. Furthermore, we discuss special properties of Chomp that can be considered winning or nearly winning. Moreover, we shed light upon the relation between the game of Nim and Chomp. Afterwards, we discuss methodical approaches for first player wins, those being: Random, Monte Carlo, Monte Carlo Tree Search, Steal-Any and Strategy-Stealing. These methods can be formed strategically, i.e., by means of winning strategies, or non-strategically. Using consecutive experiments, we investigate the performance of these methods for the first player as to discover how various methods and strategies compare in terms of performance and efficiency for first-player wins.

Contents

1 Introduction

The game of Chomp is a well-known game in the world of combinatorial game theory which has lended itself –and still lends itself– to thorough mathematical and computational analysis. It has been analyzed extensively, both mathematically and computationally, yet, room for analytical improvement remains. Moreover, algorithms can be invented and optimized to enhance the analysis of game-strategies and board-dimensions in Chomp.

In this thesis, we provide a theoretical and experimental analysis of this game that is both thorough and enhanced. We shed light upon the mathematical theory behind Chomp, as well as the computational results acquired from various experiments. Computationally, we analyze the difference in performance of various methods employed by the first player against an adversary that plays optimally. This is done over increasing board-dimensions, from small to large. Furthermore, various strategies will be embedded into these methods: a method is called a strategic method if it makes use of such strategies and is called a method otherwise. We make the first player employ different methods and strategic methods in order to win the game and we test them against each other, comparing their performances. These comparisons are first performed between the same methods and then between different methods.

In this thesis, our ultimate objective is to analyze different methods and strategic methods and to compare them to each other to see which one performs best in each category. Hence, our final aim is to answer the following research question: "How do different methods and strategic methods compare for the first player in terms of efficiency and win-loss performance for the game of Chomp?".

1.1 Thesis overview

In this thesis, we provide background and preliminary theory for Chomp in section 2. Afterwards, section 3 introduces and clarifies work related to Chomp. Subsequently, section 4 expounds on novel theory related to Chomp. Then, in section 5, an elaboration of the approach towards implementation and experimentation is provided. In section 6, the experimental results are exhibited and elaborated. Finally, section 7 contains a conclusion as well as a reference for future work. This bachelor thesis aims at enhancing the mathematical and computational analysis of the game of Chomp. It was supervised by Dr. R. van Vliet and Dr. M. van den Bergh at the Leiden Institute of Advanced Computer Science (LIACS).

2 Background & preliminary theory

The game of Chomp was first introduced formally in 1974 by the American mathematician David Gale [6]. Since then, this game has been subjected to thorough research and analysis, both mathematical and experimental, by mathematicians and computer scientists with interest in game theory.

2.1 Game and rules

In the game of Chomp, a set of $m \times n$ objects, say, chocolate blocks, is laid out in the specific shape of an $m \times n$ rectangular array, where $m, n \geq 1$. The array always starts filled with objects. In this array, an object can be specified by indicating a specific row-number i and a specific column-number j, where $0 \leq i < m$ and $0 \leq j < n$. A state is defined to be the board configuration at the moment and the set of performable moves available from it. In the initial state, the top-left object is denoted by $(0, 0)$ and the bottom-right object by $(m-1, n-1)$. The game of Chomp is a two-player game. There is an initiating player, say player A, who makes the first move and an adversary of this player, say player B, who performs a move after player A has done so. The two players alternate in play. The first player begins by selecting an arbitrary object, say (i_1, j_1). Subsequently, all objects (i, j) for which $i \geq i_1$ and $j \geq j_1$ are removed from the array. Afterwards, it is the turn of player B, who must select an arbitrary remaining object to perform a move in a similar fashion.

The game continues with the players moving alternately and the player who makes the last move loses the game. Note that if more than one object remains, a player could just remove all remaining objects in his turn, effectively causing himself to lose the game. This possibility of "misbehaviour" is irrelevant to analyze, because such "losing" moves are not reasonable for a player to perform, since a player would obviously play this game with an intent to win. For this reason, we introduce several formal assumptions and definitions to which we will adhere when discussing further rules and mechanics of the game of Chomp, as well as when committing to its analysis and when discussing the behaviour of the two Chomp-players. The assumptions and definitions are the following:

Assumption 2.1.1 (Reasonable play): Both players adhere to *reasonable play*. This means that given a game of Chomp with two players A and B, both players have the intent of winning the game and play accordingly. This effectively means that given that it is a player's turn, then that player could *never* remove all objects in the array, unless only the top-left object remains. Note that having the intent of winning is not the same as always performing winning moves; it only means that players never remove all objects by selecting object (0,0), unless that is the only object left in the array.

Corollary 2.1.2 (Forcing losses): Assuming that both players adhere to *reasonable play*, they alternately perform reasonable moves. This means that players will always aim not to remove the object located at $(0, 0)$, which in turn means that the player who does eventually take it, is forced by his opponent to take it and thereby, forced by his opponent to lose.

Corollary 2.1.3 (Last remaining object): Another implication of the assumption of reasonable play is that the last state before an end-state, that is, the state in which all objects are removed from the array, is always the state in which only the object $(0, 0)$ remains. The reason for this is as stated in Corollary 2.1.2: both players will avoid taking this object, causing it to remain as the final object as it is situated in the top-left.

Defintion 2.1.4: (Pre-terminal state): We will call the state in which only $(0, 0)$ remains the *pre-terminal state*.

Assumption 2.1.5 (Finite boards argument): Since the core objective of this thesis is to analyze the effectiveness of various strategic methods for first-player wins, we will restrict this board-dimensions to finite sizes. The reason for this is that analyzing infinite board-dimensions is computationally infeasible.

Definition 2.1.6 (Perfect play): We say that a player plays perfectly in a game G if and only if from every state that is not equivalent to the end-state, he always performs the move that has the highest probability of winning, that is, the *optimal move*. This means that if we play games where moves can be assigned certain scores in accordance with their probability of winning, the player chooses a move with the highest score. And if we play games in which each state is either winning or losing, meaning that moves are either winning or losing as well, as a winning move would be a move that puts the opponent in a losing state (and analogously for a losing move), then an optimal move would be one of the winning moves. If there are no winning moves, then an optimal move could obviously only be a losing move (do note that under reasonability, in Chomp this move still could not be $(0, 0)$, unless this is the last object left).

Definition 2.1.7 (Moves): A move is the selection of a move-option from the current state, that is, the selection of an element m of the moveset M of the current state. This means that the current state as well as the move-options that are available from it define what the possible moves are to take.

Definition 2.1.8 (Move-legality and specificity): A move is called legal from the current state if a player is allowed to perform that move from the current state and illegal otherwise. We say that a move is specific from the current state if it has a specifiable attribute that differentiates it from other moves (think of: moving a pawn versus moving a queen in Chess, distinction based on player turn, and so on).

Definition 2.1.9 (Winning and losing states): A state of the game is winning for a player who is to move from that state if that player can force the other player to lose. Otherwise, this state is a losing state.

Definition 2.1.10 (Poison): Since the person who removes object $(0, 0)$ loses, we name this object the "poisonous object" or simply, *poison*. From now on, we will go by this naming convention.

2.2 Combinatorial games

Game theory knows many branches, many of which encapsulate games that we encounter in our daily lives. An example of such a branch is the branch of zero-sum games. Well-known two-player zero-sum games are Poker, Monopoly and Stratego, in which uncertainty and chance play an important role [5]. Another branch of game theory is the branch of combinatorial games, such as Quarto, Checkers and Othello. These are the kind of games that are closer to Chomp in nature. In these games, it is typical that there is an outcome of win or loss.

In general, combinatorial games have several characteristics that stand out and distinguish it from the majority of game theoretical branches. These properties are the following [4]:

1. There are always two players, to which we may designate the numbers 1 and 2.

2. There is a finite set of game states that are reachable from the current game state.

3. Players 1 and 2 alternate in moves. They cannot perform moves simultaneously, neither do players act whenever they desire, as opposed to real-time games. The players can also not voluntarily pass their turn, as to give their opponent the turn to make a move.

4. There is a set of game rules that specifies for every state of the game that can still move to other states what the legal performable moves are for a player from that state. This means that, in terms of legality of moves, both players have the same move-options from the same state. A move is defined as a move-option from a specific state.

5. There is an initial state and there is a terminal state. Player 1 performs the first move from the initial state. This effectively means that in these games, the only difference between player 1 and 2 is that player 1 performs the first move. This, combined with the previous rule, also entails that a game is determined by a set of states and the moves inbetween these states.

   A terminal state is a state from which no move can be performed by any of the players. In that case, one of the two players must be declared the winner. Draws are not allowed in these games. Often in the cases that they do occur, ending conditions are introduced in order to eliminate the possibility of their occurrence.

   This effectively means that, in combinatorial games, every state and thereby, every move transitioning from one state to the other, is either winning or losing for a player, as no draws can occur.

6. The game must end in a finite number of moves no matter what moves are performed by the players. The set of game rules must be defined such as not to allow for any infinite sequences of moves.

7. The game must be of perfect information, that is, both players have full knowledge of the game state, as well as what moves are performable in that state. Hence, there are no "hidden" moves, in which one of the two players may perform a move that is invisible to the other, there are no hidden elements on the board and there is no private information.

8. Finally, there must be no element of chance in the performable moves. Moves that contain elements of chance, such as dice-rolling or cards-dealing are not allowed. This means that there are no "accidental" or "lucky" moves, but that instead, each move is *deliberately chosen* by a player.

Combinatorial games come in two categories. The first category is the class of partizan games. In these games, the game rules make at least one specific distinction between the moves that are performable by the players and this distinction is player-based. This means that, in terms of specificity of moves, from any game state, both players tend to have different move-options. For example, consider the game of "drawless" Tic-Tac-Toe, that contains the following ending condition: "if a terminal state is reached and no player has won the game by having three-in-a-row, then the player whose turn it is and is unable to move loses the game.". This drawless variant

of Tic-Tac-Toe is clearly partizan, as for both player 1 and 2, in all game states, the cells they can and cannot place their stone are the same for both players, meaning their move-options are the same in terms of legality, but in terms of specificity their moves are not the same, as one player is only allowed to place X-labeled stones, while the other player is only allowed to place O-labeled stones. Clearly, the moves they may perform are not similar and hence, not the same.

The second category of combinatorial games is the class of impartial games. In these games, the game rules make no distinction between the performable player moves. This means that both in terms of specificity and legality, from any game state, the moves performable by both players are the same. Consider for example the game of Quarto. In this game, there are two players who alternate in play on a 4×4-board and there are 16 pieces. The pieces have dichotomous attributes, where each piece is either tall or short, square or round, black or white and hollow or solid. Player moves consist of the other player to choose a piece and to assign it to his opponent and the opponent in turn needs to choose an empty cell on the board to place this piece on. The objective is to place four pieces in a row with similar attributes [10]. It is apparent that Quarto is an impartial game, as both players can only perform the same moves from any state of the game: they have the same pieces to assign to the other player and the other player has the same cells on the board to place the assigned piece on.

Discussing partizan and impartial games quickly raises the following question: "What about Chomp? Is this a partizan game, or is it impartial? And is it even a combinatorial game?". To analyze chomp, it is indeed required to first know the game theoretical branch it belongs to, so as to know what mathematical theorems may apply for this game. Hereunder, we provide some formal statements and proofs in which this is done.

Lemma 2.2.1 (Combinatorial property): The game of Chomp belongs to the branch of combinatorial games in game theory.

Proof: We verify that Chomp has the eight properties of combinatorial games.

1. Chomp is a two-player game, hence, the first property holds.
2. According to the game rules, the two players alternate in moves and cannot perform simultaneous or any other kind of moves. This means that the third property holds.
3. In the game rules, it is specified that there is an initial state, from which the first player performs the first move and there is a terminal state, namely the state in which no objects are left on the board. No draws can occur, as the player who takes the last remaining object, $(0, 0)$, loses the game. This corresponds to the fifth property.
4. Any game state indicates whether moves are legal: moves on remaining objects are legal, while moves on removed objects are not. Effectively, both players could only perform the same moves if they would find themselves in the same states. This means that the rules for performable moves apply to both players in the same manner. This implies that the fourth property holds.
5. According to Assumption 2.1.5, the Chomp boards to play on are finite. Let us take into account that, in accordance with the game rules, moves of players consist of taking

away a number of objects and that objects can only be removed. In any game state, it is not possible to add new objects and after object-removal, objects cannot be placed back (i.e.: moves cannot be undone). Effectively, this means that, once players perform sequences of moves, a terminating state is always reached. This terminal state would obviously be reached in a finite number of moves, no matter what moves are performed, and no infinite sequences of moves are possible, as that would require moves that could add objects or make moves undone. Effectively, this means that the sixth property holds, which in turn entails that the second property holds, as if the game ends in a finite number of moves for every possible move, because in every game state the number of performable moves is finite, then the number of possible game states must also be finite, meaning that there exists a finite set of reachable game states from the current state.

6. It indeed holds that Chomp is a game of perfect information as in any game state, both players can see the number of objects present on the board, as well as their locations, meaning that for all states, both players can see the moves that can be performed from that state. This effectively means that if it is player A's turn to move, that he as well as his opponent can see his moveset. Hence, the seventh statement holds.
7. Finally, in accordance with the game rules, there are no moves that contain elements of chance or accident, confirming the validity of the eighth statement for Chomp.

In conclusion, Chomp has all properties of a combinatorial game and hence, is a combinatorial game.

Corollary 2.2.2 (Impartial property): The game of Chomp belongs to the class of impartial games in combinatorial game theory.

Proof: We can easily verify that is an impartial game. It should hold that both in terms of specificity and legality, in any game state, the game rules make no distinction in performable player moves. Clearly this is the case, as in accordance with these rules, it holds that for any game state, there is only one moveset and therefore, both players would need to adhere to this unique moveset if they would find themselves in that state. This effectively proves that, both in specificity and legality, the game rules make no distinction in performable player moves and thereby, it is proven that Chomp is indeed an impartial game.

Definition 2.2.3 (Normal and Misère play rules): A combinatorial game is said to be played in a normal way if the last player to perform a move wins. This is called the normal play rule. Analogously, a combinatorial game is said to be played in a misère way if the last player to move loses the game. This is called the misère play rule [4].

Definition 2.2.4 (P-positions and N-positions): A state of a combinatorial game is called a P-position if and only if it is winning for the Previous player and is called an N-position if it is winning for the Next player.

Definition 2.2.5 (Ways and winning ways): In combinatorial games, a way for a player is defined as a specific sequence of moves takeable by this player to end up in an end-state.

Such a way indeed depends of what moves the other player performs. A way is said to be winning for a player if and only if it only consists of moves that, when taken, certainly allow the player to win.

Definition 2.2.6 (Winning plays): A play of a combinatorial game is said to be winning for a player if and only if this player takes a winning way during this play.

Definition 2.2.7 (Strategies): In a combinatorial game G with a set of possible states $S = \{s_1, s_2, \ldots, s_n\}$ where $n \geq 0$, a strategy is defined as an algorithm that specifies for a player the exact move he should take for any state s_i, where $0 \leq i \leq n$. Essentially, it specifies for a player the way to take along the line of play.

Definition 2.2.8 (Winning strategies): A strategy is said to be winning for the current player from the current state, if and only if it specifies for this player the exact moves to take from here onward as to end up being the winner in any possible terminal state, meaning that it allows for the current player to win no matter the moves of his adversary. That is, a winning strategy is a strategy that specifies for the current player the exact moves to take to force a win.

Corollary 2.2.9 (Loss-condition of winning strategies): In any combinatorial game, winning strategies can only exist for a player if this player can force his opponent to lose. Since combinatorial games consist of two players, forcing an opponent to lose means for the player to force a win. The specific *sequences of moves* by which this could be done from the current state would define all possible winning ways, for all possible adversarial responses. The specific *manners* in which this could be done from the current state defines the set of winning strategies for the player.

Corollary 2.2.10 (Playing perfectly): In any combinatorial game, if a specific player can force a win from the current state, then this player will always perform the exact moves to force a win when playing perfectly. That is, when playing perfectly, the player will perform a *winning play*.

Theorem 2.2.11 (Zermelo): In any two-player impartial game with alternating moves and a finite number of states in which no draws are allowed, either player 1 or player 2 has a winning strategy [12]. Here, player 1 indicates the player who started the game by performing a move from the initial state and player 2 is the other player.

Corollary 2.2.12 (Winning strategies in combinatorial games): In any combinatorial game, either player 1 or player 2 has a winning strategy.

Corollary 2.2.13 (Winning strategies in Chomp): In Chomp either player 1 or player 2 has a winning strategy. That means that this player can force a win, no matter what the opponent does.

Definition 2.2.14 (X-winning state): A state of a combinatorial game is said to be *X-winning* if and only if player X can force a win from that state onward [11]. We say that a state is *first-winning* if the first player (that is, the player who starts in this state) can force a win from this state and is *second-winning* otherwise.

Definition 2.2.15 (Winning property): A property of the game state is said to be a *winning property* for player X if and only if this property implies that the state is X-winning [11].

2.3 Impartial game properties

The strength of impartial games lies in the mathematical structures they adhere to. If it is known that a game belongs to the impartial class, then it is also known in advance what theorems hold for this game. Hereunder, we provide several mathematical properties of impartial games.

Definition 2.3.1 (Characteristic property): In impartial games following the normal play rule, P- and N-positions are defined recursively according to the following statements [4]:

1. All terminal states are P-positions.
2. From every N-position, there is at least one move to a P-position.
3. From every P-position, every move goes to an N-position.

Definition 2.3.2 (Characteristic property for Chomp): In Chomp, the rules of the characteristic property are slightly different. Clearly, the terminal state can never be a P-position, since Chomp is an impartial game that adheres to the misère play rule: the player who performs the last move is the loser, hence, terminal states could never be winning for the previous player. Thus, we will adjust the first two rules for the characteristic property in Chomp to be as follows:

1. All terminal states are N-positions.
2. From every N-position that is not terminal, there is at least one move to a P-position.

We keep the third rule the same.

Definition 2.3.3 (Directed game graphs): A directed game graph G is a pair (V, F), where V is a non-empty set of of vertices, representing game states, and F is a (possibly empty) set, of directed edges, each edge corresponding to a player move. F being the set of moves effectively means that it is a function that for every state $x \in V$ gives the states to which a player may move to from x. These states $F(x)$ are called the *followers* of x. Moreover, since V is the set of possible game states, $F(x) \subset V$ for any $x \in V$ and if $F(x)$ is empty for a given x, then this x is called the *terminal state*.

On such a graph, a combinatorial game may be played, adhering to the following rules:

1. There is an initial state, $x_0 \in V$, from which the first player, say player 1, initiates by making a move.
2. Both players, say 1 and 2, alternate in moves. And at every state x, the player whose turn it is chooses a move $y \in F(x)$.
3. The player for whom there is no y left from a state x, is declared the loser under the normal play rule and the winner under the misère play rule.

According to this definition, an infinite number of moves can be performed, which is not the case in combinatorial games. Hence, directed game graphs of combinatorial games restrict path length to a fixed maximum, say $n \in \mathbb{N}$. Every path, initiating from x_0, has a length that is smaller than or equal to n. Such graphs are called *progressively bounded*. This effectively means that in such game graphs, if V is finite, which is always the case for Chomp according to the fifth statement of the proof of Lemma 2.2.1, there are no cycles [4].

Definition 2.3.4 (Sprague-Grundy function): The Sprague-Grundy function on a directed game graph $G = (V, F)$ is a function g: $V \to \mathbb{N}$, such that:

$$g(x) = min\{n \geq 0 \text{ for which } n \neq g(y) \text{ for any } y \in F(x)\} \text{ [4].}$$

Hence, $g(x)$ becomes the minimal value that is not in the set of the values $g(y)$ of its followers y. Since the minimal excludant, mex in short, of a set of non-negative integers is defined as the smallest non-negative integer not in that set, we may simplify this notation by:

$$g(x) = mex\{g(y) \text{ where } y \in F(x)\},$$

which is used more often. It is clear that $g(x)$ is defined recursively and is self-starting, as $g(x)$ is defined in terms of $g(y)$, for all followers y of x. For terminal vertices x, $g(x) = 0$ holds, as $F(x)$ would correspond to the empty set. For non-terminal vertices x for which all their followers are terminal vertices, $g(x) = 1$ clearly holds, as $g(y) = 0$ and the minimal value not equal to it would be 1.

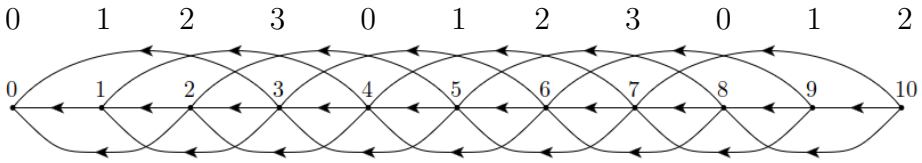

Figure 1: A directed game graph with Sprague-Grundy values for a Substraction game. In this game, states are numbers 1, 2, ..., 10. In a move, the current player may substract 1, 2 or 3 from the current numebr.

Theorem 2.3.5 (Sprague-Grundy analysis of game graphs): A directed game graph G may be analyzed either by considering the P- and N-positions or by making use of the Sprague-Grundy function [4]. This way, the Sprague-Grundy values of future states and the current state can be determined.

Proof: It is easy to show that directed game graphs can be analyzed using the Sprague-Grundy function. Suppose that we have a Sprague-Grundy function g for a directed game graph G of a game in which we play normally. Then states x for which $g(x) = 0$ are P-positions according to the characteristic property, as these would be the vertices from which a terminal state could not be reached by the current player, since Sprague-Grundy value 0 is excluded from the set of follower-values $g(y)$. Analogously, if $g(x) \neq 0$, then

those vertices x would represent the N-positions of the game. Since directed game graphs of combinatorial games are analyzable by the characteristic property, we only need to show that the Sprague-Grundy function adheres to this property, which is clearly the case as we have established how Sprague-Values immediately correspond to P- and N-positions of the game. Below, we describe the characteristic property using the Sprague-Grundy function instead of P- and N-positions:

1. If x is a terminal state, then $g(x) = 0$
2. From every state x where $g(x) \neq 0$, there must be at least one follower y such that $g(y) = 0$, since value 0 was not excluded from the set of follower-values $g(y)$.
3. From every state x where $g(x) = 0$, every follower y has $g(y) \neq 0$, since value 0 was excluded from the set of follower-values $g(y)$.

Note that, in accordance with the characteristic property for Chomp, for misère games the same rules would hold except the first two, which would need to adapted to be:

1. If x is a pre-terminal state, then $g(x) = 0$ and if x is a terminal state, then $g(x) \neq 0$.
2. From every state x that is not terminal where $g(x) \neq 0$, there must be at least one follower y that has $g(y) = 0$.

This proves that for both normal and misère games, the corresponding directed game graph can easily be analyzed using the Sprague-Grundy function.

Definition 2.3.6 (Sum of games): Suppose we have $n \geq 2$ combinatorial games whose directed game graphs are $G_i = (V_i, F_i)$ for the i-th game. And suppose that we combine these games into one large game in which a player can perform only one move in only one of the n games in a turn. Then one could combine these game-graphs into a single directed game graph $G = (V, F)$ called the *sum* of $G_1, \ldots, G_n$. Here, the set of vertices V is the Cartesian product $V = V_1 \times \cdots \times V_n$, that is the set of all n-tuples $(x_1, \ldots, x_n)$ for which $x_i \in V_i$ for all $1 \leq i \leq n$. For any vertex $x \in V$, the set of moves (or followers) $F(x)$ is defined as follows:

$$F(x) = F(x_1, \ldots, x_n) =$$

$$F_1(x_1) \times \{x_2\} \times \cdots \times \{x_n\} \cup$$

$$\{x_1\} \times F_2(x_2) \times \cdots \times \{x_n\} \cup$$

$$\vdots$$

$$\{x_1\} \times \{x_2\} \times \cdots \times \ F_n(x_n).$$

This means that, effectively, a move from $x = (x_1, \ldots, x_n)$ consists of moving exactly one of the x_i to one of its followers $y \in F_i(x_i)$. As $n \geq 2$ games are played simultaneously in this game G, we call G the *disjunctive sum*, in short, *sum* of these game graphs. This sum is denoted by:

$$G = G_1 + \cdots + G_n$$

Keep in mind that since all of the graphs G_i are progressively bounded, because they are directed game graphs, the sum G of these game graphs is progressively bounded as well. This means that the maximum number of moves from a vertex $x = (x_1, \ldots, x_n)$ is the sum of the maximum number of moves in each of the component graphs G_i. This means that even if each component game is trivial, the sum of these games can be quite complex [4].

Theorem 2.3.7 (Sprague-Grundy): For a composite combinatorial game consisting of $n \geq 2$ combinatorial games, in which a player can perform a move on only one of the games per turn, the Sprague-Grundy function can be obtained from the Sprague-Grundy functions of the component games. Suppose that for all i where $1 \leq i \leq n$, g_i is the Sprague-Grundy function of the component game G_i. Then the Sprague-Grundy function g of the game $G = G_1 + \cdots + G_n$, is defined by:

$$g(x) = g(x_1, \ldots, x_n) = g_1(x_1) \oplus \cdots \oplus g_n(x_n),$$

or simply,

$$g(x) = \bigoplus_{i=1}^{n} g_i(x_i),$$

where $\oplus$ is the XOR-operation performed on the binary representation of the Sprague-Grundy values. In case of misère play, a player loses if he performs a last move in one of the n games.

Performing the XOR-operation on the Sprague-Grundy values of the component games to obtain the total Sprague-Grundy value of the combined game is also called XOR'ing or *Nim-summing* the component game states. Essentially, the most important implication of the Sprague-Grundy theorem is that every progessively-bounded impartial game that is a component game, say G_i, in a sum of games, behaves exactly as a *pile* in the impartial game called *Nim* [4]. This in turn can simplify computations that otherwise would be complex.

Definition 2.3.8 (Do-Nothing move): In impartial games, a move from state s_1 to state s_2 is said to be a *Do-Nothing move* if s_2 is not a terminal state and all possible moves from state s_2 lead to states s_3 that were already directly accessible from state s_1 [11].

Theorem 2.3.9 (Do-Nothing property): In impartial games, if there is a Do-Nothing move from a state s_1 to some state s_2, then state s_1 is always an N-position.

Proof: Suppose that state s_1 is a P-position. Then, by definition, every move from s_1 leads to an N-position. Hence, since there is a Do-Nothing move from s_1 to s_2, s_2 must be an N-position. However, by definition of a Do-Nothing move, every move from s_2 to any state s_3 was already available from state s_1. Since every move from state s_1 leads to N-positions (due to it being a P-position), every state s_3 reachable from s_1 via this specific move must be an N-position as well. Hence, every move from s_2 leads to an N-position s_3, meaning that s_2 must be a P-position. However, s_2 was already an N-position, hence a contradiction occurs. This means that state s_1 cannot have been a P-position and instead, must have been an N-position.

Definition 2.3.10 (Do-Nothing state): In impartial games, we say that a state is a Do-Nothing state, if and only if a player can perform a Do-Nothing move from that state.

Theorem 2.3.11 (Do-Nothing moves in Chomp): In Chomp, in every rectangular $m \times n$-boards, where $m \geq 1$ and $n \geq 1$ and m and n are not both equal to 1, a Do-Nothing move can be performed. If both $m \geq 2$ and $n \geq 2$, then there is exactly one Do-Nothing move, namely $(m-1, n-1)$.

Proof: Given a rectangular $m \times n$-board; then the following statements apply:

* If $m, n \geq 1$ and m and n are not both equal to 1, then there exist several scenarios:

1. Player 1, performs the move $(m-1, n-1)$. Then we can reason that this move was indeed a Do-Nothing move. Suppose that player 2, responds by selecting an arbitrary move (x, y) from the state that was left behind by player 1.

   In Chomp, moves consist of removing all objects to the right and down from the selected object and including it. This means that the move (x, y) that player 2 selects right after player 1 has performed the move $(m-1, n-1)$, spans all positions (x_i, y_i) where $x_i \geq x$ and $y_i \geq y$, which of course includes the position $(m-1, n-1)$. Since the move player 2 performs spans over all the positions of the previous move (performed by player 1), player 1 could have performed move (x, y) directly, leaving behind the same state for player 2. In other words, $(m-1, n-1)$ is a Do-Nothing move.

   For the sake of clarity, we illustrate this argument with an example in Figure 2:

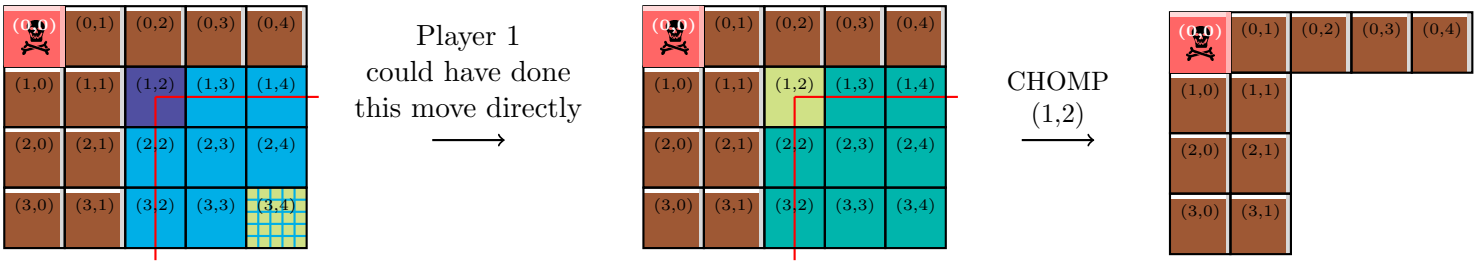

Figure 2: A visualisation of how the move of player 2 (blue) covers the previous move, performed by player 1 (green)

2. In case player 1 performs any other move than $(m-1, n-1)$, moves that can be performed can be of several kinds. We will describe each kind and provide arguments as to why these moves cannot be Do-Nothing moves:
   (a) The move where $x = 0$ and $y = 0$: this move removes all objects on the board, leading to a terminal state. Then, by definition, this is not a Do-Nothing move.
   (b) Moves where $x = 0$ and $0 < y \leq n-1$ or $y = 0$ and $0 < x \leq m-1$ or $0 < x \leq m-1$ and $0 < y \leq n-1$ where $x = y$ may occur: assume that $m, n \geq 2$. Since $(x, y) \leq (m-1, n-1)$, either $x < m-1$, $y < n-1$ or both hold. Suppose that $x < m-1$, then if $y = 0$, performing the move $(0, 1)$ after (x, y) leads to a state s_3 that is not directly obtainable from s_1. If $y > 0$, then performing $(m-1, 0)$ after (x, y) leads to a state s_3 not obtainable directly from s_1. Analogously can be reasoned if $y < n-1$ or both $x < m-1$ and $y < n-1$ hold. This means that any move other than $(m-1, n-1)$ cannot be a Do-Nothing move.

Hence, in every rectangular game state, a Do-Nothing move can be performed if this game state is not equal to the pre-terminal state and there is only one Do-Nothing move, namely $(m-1, n-1)$.

Corollary 2.3.12 (Do-Nothing property for Chomp): In the game of Chomp, every state that consists of a rectangular $m \times n$-board, where $m \geq 1$ and $n \geq 1$ and m and n are not both equal to 1, is a Do-Nothing state.

2.4 Backward induction

Backward induction is an approach that analyzes a game from the end (which can be the terminal state or another "special" state that aligns with a specific condition) all the way back to the current state and to obtain meaningful information out of this analysis. It is often used to determine whether the current state is a P- or an N-position. Using backward induction, one can prove that every possible state in a combinatorial game is either a P- or an N-position. Backward induction however has more instances of use, some of which are: determining an optimal move, determining what a good strategy is and determining which player is more favourable [4]. In this thesis, we focus primarily on two things regarding backward induction:

i). determining whether states are P- or N-positions and,

ii). by means of (i), determining a sequence of optimal moves to take for a player in order to force a win.

2.5 Partially ordered set games

In combinatorial games, there exists a branch of games called *partially ordered set games*. In these games, both players are presented a partially ordered set and moves consist of choosing a specific element in this partially ordered set to remove. Once this element has been selected, this element and all elements greater than it will be removed from the set. These games can be played in both the normal and the misère convention: in the former, the player who has no elements to remove,

loses the game and in the latter, the player who removes the last remaining element loses the game. To better understand partially ordered set games, prior understanding of partially ordered sets is required.

Definition 2.5.1 (Partially ordered set): A partially ordered set P is a set P together with a relation $\leq$ such that $\leq$ on P is reflexive, anti-symmetric and transitive, that is:

- Reflexivity: $\forall x \in P : x \leq x$.
- Anti-symmetry: $\forall x, y \in P : x \leq y \wedge y \leq x \implies x = y$.
- Transitivity: $\forall x, y, z \in P : x \leq y \wedge y \leq z \implies x \leq z$.

Definition 2.5.2 (Comparability and total order): Let P be a partially ordered set. Then different elements $x, y \in P$ are said to be comparable if $x \leq y$ or $y \leq x$ holds. A partially ordered set P is said to be a total order if all of its elements are comparable.

It is now deducible that the game of Chomp adheres to the criteria of a partially ordered set. Let C denote the set of objects (x, y), where $0 \leq x < m$ and $0 \leq y < n$. Now, define the relation $\leq$ to be: $(x, y) \leq (x', y') \iff x \leq x' \wedge y \leq y'$. Then we obtain the following:

- Reflexivity: $\forall o \in C : o \leq o$ clearly holds by definition. Thus, $\leq$ is reflexive.
- Anti-symmetry: $\forall o_1, o_2 \in C : o_1 \leq o_2 \wedge o_2 \leq o_1 \implies o_1 = o_2$. This holds due to the following: suppose that $o_1 = (x_1, y_1)$ and $o_2 = (x_2, y_2)$. Then, if $(x_1, y_1) \leq (x_2, y_2)$ then $x_1 \leq x_2$ and $y_1 \leq y_2$ and if $(x_2, y_2) \leq (x_1, y_1)$ then $x_2 \leq x_1$ and $y_2 \leq y_1$. Combining these two statements must imply that $x_1 \leq x_2 \wedge x_2 \leq x_1$ and hence $x_1 = x_2$ and $y_1 \leq y_2 \wedge y_2 \leq y_1$ and hence $y_1 = y_2$ thus, $(x_1, y_1) = (x_2, y_2)$, meaning that $\leq$ is anti-symmetric.
- Transitivity: $\forall o_1, o_2, o_3 \in C : o_1 \leq o_2 \wedge o_2 \leq o_3 \implies o_1 \leq o_3$. suppose that $o_1 = (x_1, y_1)$, $o_2 = (x_2, y_2)$ and $o_3 = (x_3, y_3)$. Then, if $(x_1, y_1) \leq (x_2, y_2)$ and $(x_2, y_2) \leq (x_3, y_3)$, then $x_1 \leq x_2 \wedge y_1 \leq y_2$ and $x_2 \leq x_3 \wedge y_2 \leq y_3$. Since this holds, it immediately follows that $x_1 \leq x_3 \wedge y_1 \leq y_3$. Thus, $\leq$ is transitive.

Hence, C forms a partially ordered set. For this reason and because Chomp adheres to the description above, Chomp is considered to be a partially ordered set game.

2.6 Well-known strategies

There exist several well-known strategies that allow for mathematical and computational simplification of Chomp-board analysis. Using these strategies, one can easily observe that a player is able to win the game from a certain state. This allows for game states to be analyzed much quicker. Moreover, such strategies are useful for a player: not only do they reduce the complexity of plays significantly, but they also allow for the player to win, regardless of adversarial moves. Below, we provide a list of well-known winning strategies [6], which we will also incorporate in our implementation.

- **Strategy 1:** In the case of a $2 \times n$-board, a player can win from this state by selecting the object $(1, n-1)$ and responding as follows to the moves of the other player: if the other player removes $(0,0)$ then that is an immediate loss and otherwise, whatever move the other player makes, the response should be the move that leaves behind a state in which there is precisely one more object in the first row than in the second row. Such moves are clearly possible, because if player 2 removes an object from row 0, a $2 \times j$-rectangle emerges from which player 1 can remove $(1, j-1)$ and if player 2 removes an object from row 1, say $(1, j)$, player 1 can remove $(0, j+1)$. It is quite apparent that this is a winning strategy from this state. Analogously, in an $m \times 2$-board, one follows the exact same strategy, except that the moves to perform are now upon columns instead of rows: the player whose turn it is should remove the object $(m-1, 1)$ and to any adversarial response should perform a move such that the first column always contains exactly one more object than the second, until this player eventually wins.

- **Strategy 2:** In square $m \times m$-boards, the player whose turn it is can win from this state by selecting the object $(1,1)$, leaving only objects in row 0 and column 0. From there onward, this player should mirror the moves of his opponent: this means that if the opponent performs the move $(0, j)$ then the player should perform the move $(j, 0)$ and analogous for if the opponent performs the move $(i, 0)$. It is clear that this is a winning strategy for the player, as since the board is square, the number of objects in the first column and first row are equal, meaning that every move the opponent performs in this column (or row) can be performed symmetrically in the row (or column). Eventually, the opponent will be forced to take the last object.

- **Strategy 3:** In the case of $1 \times n$-boards or $m \times 1$-boards with $m > 1$ and $n > 1$, the winning strategy for the current player is trivial. The current player should make a move such that only the poisonous object remains. To do so, this player must remove $(0,1)$ in the case of a $1 \times n$-board and $(1,0)$ in the case of a $m \times 1$-board, in order to force a win.

What is noteworthy about these winning strategies is that they are only applicable in game states that adhere to certain described properties. Since these are winning strategies, this means that the following properties are winning properties for the current player: $2 \times n$-boards, $m \times 2$-boards, square $m \times m$-boards, $1 \times n$-boards and $m \times 1$-boards. Any player who is currently in these states can force a win by complying with the required winning strategy. Effectively this (in accordance with Definiton 2.2.15) means that these states are winning for the player who is to move from them.

2.6.1 Strategy stealing

Most notably, there is an interesting property that is applicable in the game of Chomp and is widely used nowadays in many other impartial games. This property is the possibility for a player to "steal" the strategy of the opponent. This is called strategy stealing. Strategy stealing is especially useful in a situation in which the opponent seems to have a winning strategy. Below, we provide a small theorem and proof for the strategy stealing argument in the game of Chomp.

Theorem 2.6.1.1 (Strategy stealing argument): For any rectangular $m \times n$ Chomp-board where $m \geq 1$ and $n \geq 1$ and m and n are not both equal to 1, the game is winning for the

current player, say player 1. That is, player 1 has a winning strategy. This means that player 1 can force player 2 to lose the game.

Proof: Recall that according to Corollary 2.2.13, either player 1 or player 2 must have a winning strategy in Chomp. Suppose that player 2 has a winning strategy, say S. Then, for every move player 1 would perform, player 2 can secure a win by forcing it using his strategy S: every time, perform the exact move (x, y) that S specifies. Now suppose that player 1 starts by performing the move $(m-1, n-1)$; then the following occurs:

Since player 2 has a winning strategy, the selected move $(m-1, n-1)$ must be losing for player 1. In that case, there must be a response, say (x, y), by player 2 to perform that is winning and this must be repeatedly the case, that is, for all future player 1 moves, player 2 will always leave behind a P-position. In Theorem 2.3.10, we have stated that in rectangular states, the move $(m-1, n-1)$ is a Do-Nothing move by player 1, meaning that player 1 could also perform any move that player 2 can perform as a response to the move $(m-1, n-1)$. We could say that player 1 could *steal* this move from player 2, due to the Do-Nothing property Chomp adheres to. Now, consider that in Chomp, its impartiality and game rules specify that the performable move-options are player-independent: when either one of the players performs a specific move, the next state obtained is the same as whenever the other player had performed that move; the only difference lies in whose turn it is (which is independent of the state). This means that the strategy S could equally well be employed by player 1. Now, a contradiction comes to rise: if player 2 had a winning strategy and could force a win by (i) performing the move (x, y) to leave behind a P-position and (ii) performing moves that S specifies afterwards, then so could player 1 force a win by performing move (x, y) directly and afterwards, performing moves that comply with S. It is however, not possible for both players to have a winning strategy from the same state. Hence, our assumption that player 2 has a winning strategy must be wrong. In other words, by stealing the strategy of player 2, player 1 can force player 2 to lose. This gives birth to the name *strategy stealing*, accurately describing this approach.

Corollary 2.6.1.2 (First-winning): The initial state in the game of Chomp is first-winning if and only if it is not the pre-terminal state $(0, 0)$.

Proof: In accordance with the game rules, the initial state is always square or rectangular in Chomp. This means that by Theorem 2.6.1.1, the initial state is winning for the player whose turn it is and this player has a winning strategy, except for the board-state where $m = 1$ and $n = 1$, that is, the pre-terminal state. In the initial state, the player who is to make a move is always the first player. Therefore, if the initial state is equal to the pre-terminal state, it cannot be first-winning and otherwise, this state must be first-winning.

This result cannot come a a surprise. By Theorem 2.3.9, each Do-Nothing state is an N-position. By Corollary 2.3.12, the initial state of Chomp for $m \times n$-boards with m and n not both equal to 1 is a Do-Nothing state. In this thesis, we will analyze the effectivity of strategy stealing for first-player wins and we will compare this to that of various other strategies and strategic methods.

2.7 Slicing Chomp

Slicing Chomp is a novel variant of the game of Chomp. In this variant, the exact game rules of Chomp apply, with one exception. That is, the moveset is extended, with the emergence of a new move-option: slicing. In Slicing Chomp, the players are presented with a shared set of tokens that may contain no more elements than the smallest dimension of the board $min(m, n)$. The tokens are visible to both players and available for the current player to move. A player whose turn it is may choose to either perform a regular chomp-bite or a so-called chomp-slice. Such a slice consists of selecting an object (x, y) and instead of removing all objects to the right and downward, this entire piece is "sliced off", meaning that it remains in the game state but is separated from the main board that contains the poison. To perform such a slice, a token must be available and subsequently, this token is removed. After one or omre slices have been performed, the players have multiple boards to choose from to perform either a bite or a slice. The current player must choose one of those boards. Effectively, slices turn the large game G into a sum of several smaller games $G_1, \ldots, G_n$ where $n > 0$, as a player can move only on one of the boards per turn. This means that by using the Sprague-Grundy theorem, if we know the Sprague-Grundy values $g(x_1), \ldots, g(x_n)$ for the current states x_1 of G_1, x_2 of G_2 and so on, we can determine the Sprague-Grundy value $g(x)$ of the combined state x of game G. Moreover, sliced boards tend to be smaller and hence, their Sprague-Grundy values are simpler to determine.

2.7.1 Strategy stealing cancellation

In Slicing Chomp, it is evident that the strategy stealing argument eventually no longer holds in theory. Let the player whose turn it is be the first player and the other player be the second. Then, given a rectangular $m \times n$-board, performing the move $(m-1, n-1)$ by biting is no longer a Do-Nothing move. Suppose that the first player indeed performs this bite. Then the second player could respond from this state by slicing off a piece from an object (x, y). The state that emerges however is not a state that was directly reachable by the first player from the initial state, hence, strategy stealing does not hold anymore.

3 Related Work

There are several games that are known to be related to the game of Chomp. Among the most well-known are: Schuh's Game of Divisors, Hackenbush and most notably, the game of Nim. Each game pertains to Gale's game of Chomp in its own way. Below, we describe each of these games as well as their rules and mechanics.

3.1 Schuh's Game of Divisors

In 1951, the Dutch mathematician Frederik Schuh introduced the Game of Divisors [13]. It is a mathematical game, that is, a game whose strategies and rules are determined by precise mathematical axioms, based on a specific divisor-number, say N. It is a combinatorial game of two players. In this game, the prescribed rules are as follows:

The game starts by designating two opposing players to agree on an arbitrary positive integer, say N, which must be larger than 1. After that, all divisors of the integer N are listed, including the integer 1 and N itself. Thereafter, one player can initiate gameplay by performing a legal move. A legal move consists of selecting one of the remaining divisors of N, say d. The move crosses out the chosen divisor d as well as every other divisor of d. The two players alternate in moves and can choose any divisor in accordance with their liking. Finally, the player who is forced to take the divisor N itself loses the game.

3.1.1 Relation with Chomp

Let us start by describing an intriguing property of the Game of Divisors: the property that relates this game to Chomp. Just as how the game of Chomp has a grid that can be represented as a partially ordered set, the Game of Divisors also forms a partially ordered set, namely under divisibility. Suppose that we have a divisor d of a number N that is greater than 1, as to exclude 1 for initial gameplay. Define the binary relation $\leq$ over the positive natural numbers by: $x \leq y \iff x$ divides y (meaning that y is a multiple of x). Then it holds that divisibility of integers closes under the properties of reflexivity, anti-symmetry and transitivity, effectively proving that the set of divisors forms a partially ordered set. We will provide a small proof for this statement.

Let $D(N)$ be the set of positive divisor of the Game of Divisors with chosen integer N: $D(N) = \{d \in \mathbb{N} : d \text{ is an integer} \wedge d \text{ divides } N\}$. Then, for any integers x and y, the following properties hold:

- Reflexivity: $\forall x : x$ divides x, that is, $x \leq x$; this holds as every integer divided by itself equals 1.
- Anti-symmetry: $\forall x, y : x$ divides $y \wedge y$ divides $x \implies x = y$; this holds for the following reason: by definition of division, if x divides y, then there exist a specific integer k, such that $y = k \cdot x$. Similarly, if y divides x then there exists an integer ℓ such that $x = \ell \cdot y$. Combining these two statements, we get $x = \ell \cdot y = \ell \cdot k \cdot x$ which means that, because k and ℓ are integers, $k = \ell = 1$ and hence, $x = y$.
- Transitivity: $\forall x, y, z : x$ divides $y \wedge y$ divides $z \implies x$ divides z; this property holds due to the following. Suppose that x divides $y \wedge y$ divides z holds. Then, by definition of division, it follows that there exist a specific integer k, such that $y = k \cdot x$ and there exists an integer ℓ such that $z = \ell \cdot y$. We can once again combine the two statements to obtain: $z = \ell \cdot y = \ell \cdot k \cdot x$, meaning that x divides z, as there exists an integer m such that $m \cdot x = z$, namely $m = k \cdot \ell$.

From the above proof, we can conclude that the set of positive divisors $D(N)$ of the Game of Divisors with the binary relation $\leq$ is a partially ordered set whose least element is 1, because $\forall x \in D(N) : 1 \leq x$, and greatest element is N, as $\forall x \in D(N) : x \leq N$.

Originally, the Game of Divisors would be played on an (unstructured) set of numbers or numbered objects in which one would select a specific divisor d,, subsequently removing all of its divisors from the array. As we have just observed, the set of divisors N forms a partially ordered set. The objects in Chomp also form a partially ordered set, as stated in Section 2.5. As a result, we can mimic the game of Divisors for certain numbers N by an instance of the game of Chomp,

by carefully arraning the divisors of N in an $m \times n$-array.

We can best illustrate this with an example:
Suppose that $N = 432$, with prime-factorisation $2^4 \times 3^3$. Since any divisor d of $2^4 \times 3^3$ satisfies $d = 2^i \times 3^j$ with $0 \leq i \leq 4$ and $0 \leq j \leq 3$, the Game of Divisors for $N = 432$ has prime-factorisation corresponds to a 4×5 Chomp-board. In this Chomp-board, each square contains one such divisor. A player can choose whichever divisor d he wants and subsequently, all of its divisors will be removed, effectively mimicking a Chomp-bite due to the partially ordered set property. In figure 3, we illustrate this behaviour for a specific divisor.

Figure 3: A visualisation of the correspondence between Divisor moves and Chomp-bites

Indeed, for the game of Divisors to correspond directly to two-dimensional Chomp-boards, the number N must satisfy a specific property. If its prime factors adhere to the formula $P_1^k \times P_2^\ell : P_1, P_2 \in \mathbb{N}^+ \wedge k, l \in \mathbb{N}$, a rectangular $m \times n$ Divisor-board can be constructed, where $m = k + 1$ and $n = \ell + 1$, which corresponds to a Chomp-board of dimensions $m \times n$. [7].

The correspondence between Chomp and the Game of Divisors is not only restricted to two-dimensional Chomp. Essentially, it has been proven that, in general, for any k-dimensional Chomp game, a Divisor game exists where N has precisely k prime divisors [7]. Given a k-dimensional Chomp-board $n_1 \times n_2 \times \cdots \times n_k$, the corresponding N should adhere to $N = P_1^{n_1-1} \times P_2^{n_2-1} \times \cdots \times P_k^{n_k-1}$ for different prime numbers $P_1, P_2, \ldots, P_k$. An example of this is Cubic Chomp over a $5 \times 3 \times 2$-field. In that case, a corresponding Divisor game could be generated by using $N = 720$, as the factors of 720 are $2^4 \times 3^2 \times 5^1$. What stands out about Schuh's Game of Divisors is that it was invented over two decades before David Gale introduced his game of Chomp, yet it turned out to be perfectly isomorphic to its successor [7].

3.2 Bouton's game of Nim

In 1901, the game of Nim was introduced by the American mathematician Charles Bouton [2]. Nim is a two-player combinatorial game in which the initial state of the game consists of three piles. These piles consist of objects, called counters, and are of arbitrary size, say $S_i \in \mathbb{N}$ per pile i. The piles are not allowed to be of equal size at the beginning. From the initial state onward, arbitrary moves can be made by the players. A legal move consists of selecting one of the piles and taking from it the desired number of counters, the least number to take being one. Players are not allowed to take counters from multiple piles. The players alternate in play and the player who takes the last remaining counter(s) from the last remaining pile wins. The game of Nim can also be played in a misère way. In that case, the player who removes the last remaining counter(s) loses the game.

3.2.1 The safe combination property of Nim

Inherent to Nim is the property of *safe combinations*. The importance of this property is that it relates itself directly to the aforementioned disjunctive sums and this property is what caused such sums to attain the name of Nim-sums. Evidently, Nim-sums lie central to the Sprague-Grundy theorem and using Nim-sums, we can determine whether a player can win or not assuming perfect play in states of Slicing Chomp where there are split pieces. A safe combination is a specific set of numbers, each representing a pile that can be left over by a player in the game state for his adversary. Suppose that we have a player A that has an adversary B. Then the characteristic of such a safe combination of numbers is that if player A leaves one behind, then his adversary B, cannot win the game if player A proceeds to play perfectly. This means that if player A is to leave a safe combination behind and does not make mistakes in future play, he can leave safe combinations behind for any adversarial move and the adversary B can never leave behind a safe combination and is bound to lose the game, as player A would eventually take the last counter [2].

A safe combination is determined by taking the number of counters in each pile and translating them into binary notation. These binary numbers are stacked onto each other, forming three horizontal lines whose units correspond in their vertical columns. The set of numbers is then summed without carry. In case the sum of *each* column is 2 or 0, then the set of numbers forms a safe combination [2].

Moreover, if two numbers are given, then the third number is always uniquely determined to form a safe combination, as it should be the number that ensures the column-sums end up being either 2 or 0. The uniqueness-requirement of the third number is what makes an adversary unable to leave a safe combination after a player has left one. If a player has left a safe combination behind, then the adversary, who needs to take an arbitrary number of counters from one of the piles, is bound to change one of these three numbers. Since these numbers are uniquely related to each other, changing a number would cause the safe combination to be broken, because the other two numbers depended on the changed number to form their safe combination. As a result, the adversary cannot leave behind a safe combination if the previous player did so. Analogously, if the adversary cannot leave behind a safe combination, the next player can leave such a combination behind by choosing a pile and removing the number of counters specifically as to obtain a pile-size equal to the required unique number that causes all columns to be summed to 2 or 0. In Figure 4, we demonstrate how safe combinations can be left behind by a player (and how they are broken by his adversary).

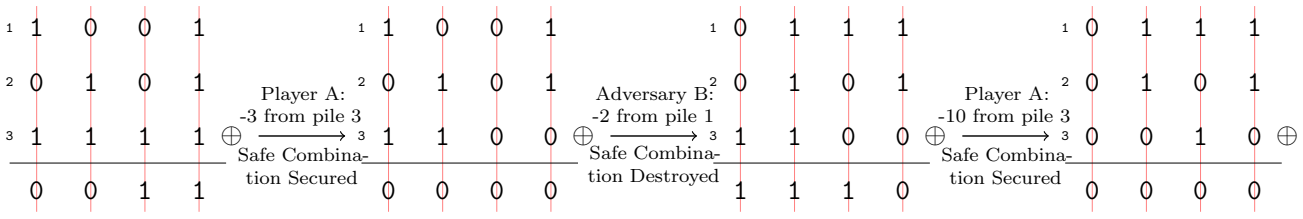

Figure 4: Leaving behind a safe combination which is forcibly broken

As said, such sums of numbers are called Nim-sums and using these properties of the Nim-sum, one can determine whether states of the game are winning for the previous or for the next player. If the current state consists of piles whose Nim-sum forms a safe-combination, then this state is winning for the previous player, that is, a P-position. This is demonstrated in figure 4, where the utmost right state represents a P-position. Analogously, if a state does not constitute a safe combination, then this state is winning for the next player, that is, an N-position. In P-positions, the previous player has left behind a safe combination after his move and in an N-position, the player to move is able to do so. This has one important implication for initial states of Nim: an initial state is already either a safe combination or not. This means that in the game of Nim, assuming perfect play, we can easily determine from the initial state whether the first player is going to win or lose the game.

3.2.2 Misère plays in k-pile Nim

The game of Nim is often played in a k-pile variant. In this variant, the game starts with k piles, where $k \in \mathbb{N}^+$ It is clear that the property of uniqueness can be extended to k-pile Nim: if $k-1$ pile sizes are given, then the required size of pile k for a safe combination can easily be determined as done for the 3-pile variant. This allows us to analyze ordinary and misère plays of k-pile Nim:

- **1-pile:** In case we have one pile, a normal play could be won by the first player by taking the entire pile and a misère play could be won by taking the entire pile except for one counter. In the case of a misère play, if there is only one counter, the first player obviously loses.

- **2-pile:** In case we have two piles, for a normal play, the P-positions are those for which the two piles have an equal number of counters, such as (1,1), (2,2) and so on, as the next player would be forced to create two piles with unequal numbers of counters when moving from these states [4]. The other player could then always make a move to leave behind both piles with an equal number of counters. Note that in the terminal state (0,0) both piles have an equal number of counters, hence, perfect play assumed, the second player could eventually make a move towards the terminal state, that being a winning move. Keep in mind that the P-positions being the states in which the piles have an equal number of counters is also in line with the property of safe combinations; given one pile of a specific size s, the unique pile-size u, for the second pile can be determined using the Nim-sum. To obtain a Nim-sum of 0 for the given s, pile-size u of the other pile can only be equal to s, as otherwise $s \oplus u$ could not result in 0.

  In the case of a misère play, a player could use the same strategy as in a normal play, as enforcing the piles to have an equal counter-amount doesn't immediately make the player lose. Hence, the P-positions remain the same with two exceptions, the first of them being that (1,1) is no longer a P-position, as the player who leaves this state behind will lose the game, and the second exception being that the terminal state (0,0) is the state that is left behind by the losing player. For this reason, (0,1) and (1,0) are the P-positions that replace (1,1), as the player who is able to leave these states behind will win the game. The misère strategy for a player, hence, is similar to the strategy in normal play, the only difference being that if one pile is of size 1, the other pile should be removed entirely by the player to be of size 0 instead of 1. And if one pile is of size 0 and the other is of size $n \geq 2$, then the pile of size n should be reduced to size 1.

- **k-pile:** We have already explained how a player could beat his adversary in normal play in the case of 3-pile Nim: the Nim-sum could be used to determine the unique amount of counters for a pile each time to leave behind a safe combination, forcing the adversary to lose eventually. This same strategy can be applied for k-pile Nim where $k \geq 3$, as we stated that the uniqueness property, and thereby the possibility of leaving safe combinations behind, holds for k-pile Nim as well.

  In case of a misère play, a player could play as he would in a normal play, *as long as* there are at least two piles with more than one counter. Whenever the adversary makes the move that leaves behind just one pile with more than one counter and other piles with exactly one or no counters, the player should reduce the last pile with more than one counter to either one or zero, whichever leaves an odd number of piles of size one remaining [4]. Since there does not exist a safe combination with exactly one number greater than 1, this strategy never allows the player to leave exactly one pile of size greater than one behind. Hence, after a certain number of moves, the game will eventually force the adversary to take away the final counter, as an odd number of piles with one counter were left behind.

3.2.3 The Sprague-Grundy theorem and Nim

As we have stated in the Sprague-Grundy theorem, the implication of Nim-pile equivalence is important. The reason for this is that when we are dealing with impartial games that have complex game states, such as states that allow for many move-options, it becomes very complex to analyze and computationally expensive. Nim-pile equivalence makes the analysis of complex games feasible. To the end of simplifying the analysis of complex impartial games using Nim-equivalence, we introduce a statement:

> **Theorem 3.2.3.1 (Nim correspondence):** Every impartial game that is progressively bounded corresponds to the game of Nim [4]. If no counters can be restored from or added to a Nim-pile, then this game correspond directly to Nim. Otherwise, it corresponds to a variant of Nim.

3.2.4 Miscellaneous variants of Nim

There exist many variants of Nim, two of which are Moore's Nim_k and Wythoff's game. In Moore's Nim_k, there are two players and the game starts with one large pile, from which the first player takes any desired number of counters and separates them at will into any number of piles, say n. Players draw alternately from these n piles, where players may remove any desired number of counters from any number of piles not exceeding k [9]. The second player performs the first draw, in which he must remove at least one counter from at least one pile (and at most k piles). From each of the desired piles, the player can remove any desired number of counters independent of the other piles. In Wythoff's game, two piles of counters are provided, each of an arbitrary number. Two players alternately take either an arbitrary number of counters from one pile, or an equal amount of counters from both piles. The player who takes the last counter(s) wins the game [15].

3.3 Conway's game of Hackenbush

Hackenbush is a game that was invented by John Horton Conway and was formally introduced in 1982, in *Winning Ways for Your Mathematical Plays*. Multiple variants of this game exist, the impartial one of which is called Green Hackenbush [1]. The game state of Green Hackenbush consists of one or more graphs and a special vertex, called the ground. Every vertex is attached by some path to this special vertex. As a result, there can be multiple graphs connected to the ground. The game is played by two players alternating in moves. A move consists of selecting an edge of one of these graphs, effectively removing that edge and all edges that are no longer connected to the ground via a path as well. Non-branching paths are called stalks and it is clear that a game state with multiple stalks corresponds to a game state of Nim with multiple piles, where the number of edges for each stalk correspond to the number of counters per pile. Paths often do branch however and such branches can be dealt with using the Colon principle: when branches come together at a vertex, they may be replaced by a stalk whose length equals the Nim-sum of the branches. In other words, using the Nim-sum repeatedly, branching trees can always be turned into single stalks whose lengths equal the recursive Nim-sum over the branches [4]. This implies that branching trees correspond to Nim-piles as well. The Colon principle is used to find the Sprague-Grundy values of a tree and Nim-summing these values gives us the Sprague-Grundy value of the state. Just like Chomp, Green Hackenbush is an impartial game to which the Sprague-Grundy theorem applies and in both games, states have their corresponding Sprague-Grundy values. Note that in Green Hackenbush, cycles may also exist in graphs. As a consequence, the Fusion principle was invented, which states that neighbouring vertices may be brought together into a loop that in its turn can be replaced by a leaf (as a loop is but an edge joining a vertex to itself). This process is called "fusion" and as a result of the Fusion principle, vertices in any cycle may be fused together without adapting the Sprague-Grundy value of the graph [4].

4 Theory

4.1 Winning properties in Chomp

Due to the earlier described winning properties for Chomp (Section 2.6), we know that some states are winning states for the player whose turn it is. We can however further generalize several winning properties that are applicable in the game of Chomp for general $m \times n$-boards instead of specific board-dimensions. Hereunder, we provide a brief outline of these winning properties.

- According to strategy 1, $2 \times n$-boards (and $m \times 2$-boards) are winning states. Taking this strategy into account however, we can further extend upon winning properties in Chomp. In any staircase-shaped board in which the first row is larger than the second by exactly one object and there are more than two rows, the player to move from this state can leave behind a P-position by removing the object $(2, 0)$. Of course for columns there is an analogous argument ($(0, 2)$ would have to be removed). This means that any staircase-shaped board in which the second row (or column) is shorter than the first by one object and with at least three rows (or columns) constitutes a winning property for the current player.

- Another winning state is the state in which there are exactly two rows (or columns) where the first contains at least two objects more than the second. If the second row (or column) is

of size k, then by selecting the move $(k+1, 0)$, a P-position is left behind and strategy 1 can be applied from there onwards.

- Strategy 2 says that squareness of boards is a winning property. However, it is not necessarily the squareness that makes it a winning property, rather, it is the property of the first row and first column being equal in size, where at least the object $(1, 1)$ is also present. In that case, this strategy can be applied as well and this is the winning property behind squareness.

  Moreover, L-shaped boards, boards with exactly one row and one column, in which either the first row is larger than the first column or vice versa, are winning states for the player who is to move from them: this player can remove a number of counters from the larger part such that the first row and column are equal in size. Then, this player can continue by mirroring the moves of the other player.

  In short, this means that game states in which the first row is equal to the first column in size and $(1, 1)$ is present and game states that are L-shaped as described above are both winning states.

It is clear that it would be clever for a player to apply these strategies if the current state adheres to one of the winning properties described above. Therefore, we provide a definition for playing cleverly.

Definition 4.1.1 (Clever play): A player plays *cleverly* if he makes use of the winning strategies and properties of a game so as to win the game.

4.2 The steal-any approach

We introduce the steal-any approach as a novel winning way for the first player. The steal-any approach is an algorithm that assigns an arbitrary winning strategy to this player given that this player starts without a winning strategy.

It does so by selecting a winning move at random from all winning game states in which the first player has to perform a move. While a winning play is being performed by this player, the strategy corresponds to an arbitrary winning strategy. When the game terminates, the exact winning strategy that has been performed can be traced back.

4.3 The Nim-Chomp relation

According to the Sprague-Grundy theorem, the game of Chomp has a corresponding Sprague-Grundy value, which would be $g(x)$ if x represents the state the game is currently in. In accordance with the Sprague-Grundy theorem the game of Chomp behaves like a single nim-pile, with a pile of size $g(x)$, indicating that there is a relation between Nim and Chomp. For the sake of mathematical exactness and computational simplicity, we are interested in clarifying the relation between the game of Chomp and the game of Nim.

First of all, note that in the game of Nim, a pile clearly represents a partially ordered set that is totally ordered: one can number the counters from bottom to top and a counter cannot be

removed from this set without removing all counters that come after it in order; for any counter i, all counters j stacked on it depend on it in the sense that removing i from the pile removes all such j as well. For the sake of simplicity, we denote a Nim-pile of size n by the totally ordered set $P_i = \{0, \ldots, n-1\}$ where i is the pile-index. A game of Nim consisting of r piles then becomes: $N = (P_1, \ldots, P_r)$.

4.3.1 From Nim to Chomp

There is a remarkable relation between the game of Nim and the game of Chomp. Let us consider the most trivial variant of Nim, i.e.: the game of Nim with only a single pile. We call this variant *single-pile* Nim and it may have an arbitrary number of counters m in its initial state. Let $P = \{0, \ldots, m-1\}$ be the partially ordered set that corresponds to this pile. Then this game can be turned into Chomp as follows.

The number of elements in P becomes the number of rows that will be obtained after spanning these counters over the second dimension. Now, let P_{span} be a single Nim-pile that contains exactly the number of counters corresponding to a certain width to span to, say n. This width clearly corresponds to the number of columns that will be created by spanning P. From hereon, we can span single-pile Nim P over two dimensions, to create a two-dimensional Nim-board visualisable in a grid. Let *Board-Nim* be the new game of Nim spanned over two dimensions; then this is done as follows:

$$\text{Board-Nim} = P \times P_{\text{span}} = \{(x, y) : x \in P \wedge y \in P_{\text{span}}\}$$

Now, we can visualize the given Board-Nim game in a Nim-grid; note that the axes of the Nim-grid are also defined in terms of counters (elements of the partially ordered sets of the piles) instead of real numbers. Below, we provide this visualization:

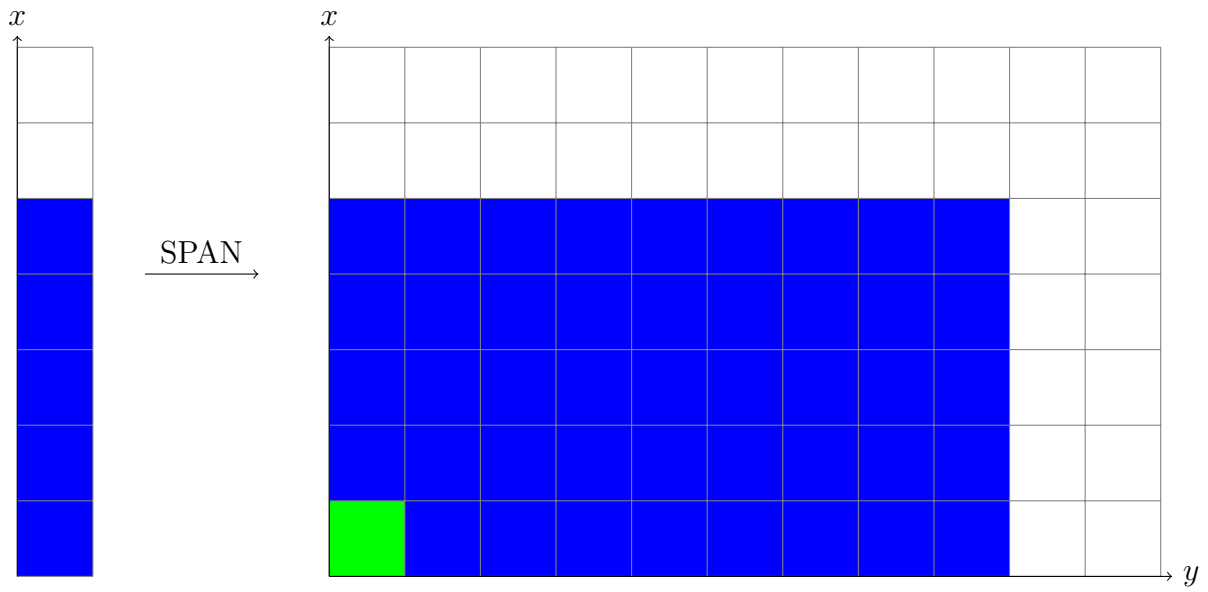

Figure 5: Spanning single-pile Nim over two dimensions to obtain Board-Nim

For the sake of clarity, we will refer to the counters in Board-Nim as objects from hereon. In Board-Nim, a move now consists of selecting a specific available object (x, y) where $0 \leq x < m$ and $0 \leq y < n$. Subsequently, all objects (x_c, y_c) for which $x_c \geq x$ and $y_c \geq y$ are removed from the board. In other words, moves in Board-Nim tend to be the same as moves in the game of Chomp.

However, there remains one subtle difference. Board-Nim is currently following the normal play convention, meaning that the player who removes the last object wins. However, in Chomp that is not the case. The misère play convention states that the player who removes the last remaining object loses the game. Since Board-Nim is born out of Nim and Nim is a game in which both conventions can be applied, the misère play convention is applicable in Board-Nim as well, with $(0, 0)$ as terminal object. Now, the game of Board-Nim corresponds exactly to the game of Chomp, as the moves are identical and both have $(0, 0)$ as their poisonous object.

4.3.2 From Chomp to Nim

Similar to how the game of Nim relates to Chomp, the game of Chomp also relates to Nim. When using backward induction upon a Chomp-board, one can lay out the elements from the pre-terminal state onward in canonical order: $(0, 0), (0, 1), (1, 0), (0, 2)(1, 1)(2, 0), (0, 3), \ldots$ and so on. The reason why this is useful is because quickly, a pattern of transformation emerges: if we choose to represent each row as Nim-pile, in which we map each row-object index-wise, meaning that the index of every row-object corresponds to a counter of the same index in the corresponding Nim-pile, then we can map the game of Chomp to a special variant of the game of Nim with multiple piles. We call this tranformation from rows to piles *fold*. For the sake of clarity, we provide a visual example:

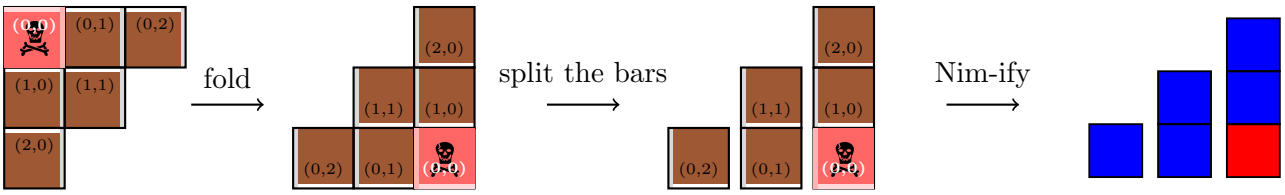

Figure 6: Transforming Chomp into multi-pile Nim

Similar to other variants of Nim, this variant follows slightly different game rules. Let $N = (P_1, \ldots, P_r)$ be the Nim-game obtained by the translation from Chomp to Nim, where r is the number of piles, $P_i = \{0, \ldots, n_i - 1\}$ is an arbitrary pile with pile-index i for which $1 \leq i \leq r$ and $n_i \geq 0$ is the size of pile P_i. Then the game rules for this variant are as follows:

1. The game starts with all r piles being filled, that is: they are all of equal size n. The pile that contains the poisonous counter is P_r.

2. Let us represent the size of a certain pile i by $||P_i||$. Then the piles in this game have the following property: given two adjacent piles P_i and P_{i+1}, where P_i is located to the left of P_{i+1}, then we must have $||P_i|| \leq ||P_{i+1}||$. This means that the piles themselves are all comparable based on size. We call this property of piles the *neighbour-property*.

3. In the initial state, the first player performs a move by choosing an arbitrary pile P_i and choosing the number of counters to remove from it. Players do so alternately, until the poisonous counter is removed.

4. Moves in this variant become slightly different, due to the neighbour-property of piles. While a move still consists of choosing an arbitrary pile and selecting a counter from it to remove as in ordinary Nim, the effect of these moves becomes different: if a move upon an arbitrary pile P_{i+1} is performed such that this pile has become smaller than its left neighbour P_i, then the size P_i will be reduced to become equal to that of P_{i+1}. And if the size of P_i becomes smaller than the size of its left neighbour after this size-reduction, then P_{i-1} must be reduced as well to equal size, and so on. On the other hand, if the size of P_{i+1} is reduced but remains larger than or equal to its left neighbour, then its left neighbour –and all of its neighbours– remains untouched. This in turn has two consequences for the behaviour of player moves:

   i). Let N be an integer array where $\mathrm{N}[i]$ represents the size of pile i. Then, performing a move m on a pile P_j in N, where $m(P_j)$ represents the number of counters to remove from P_j, results in pile-sizes $\mathrm{N}'[i]$ satisfying the following recurrence relation:

$$\mathrm{N}'[i] = \begin{cases} \mathrm{N}[i] & : i > j \\ \mathrm{N}[j] - m(P_j) & : i = j \\ min(\mathrm{N}[i], \mathrm{N}'[i+1]) & : i < j \end{cases} \tag{1}$$

   ii). When moves are performed on piles P_i of size n_i where $k \geq 1$ counters are removed, then in this move, piles are dependent if $n_i - k < n_{i-1}$. In this case, n_{i-1} must also be reduced. However, if $n_i - k \geq n_{i-1}$, then this move is completely independent from the other piles and is only performed on P_i.

This special variant of Nim that is obtained out of folding Chomp will be called $\mathrm{Nim}_{\mathrm{CHOMP}}$ and has the properties described above.

4.3.3 A remarkable property

What is most intriguing about the game of Chomp is that it is effectively a game that originates from the game of Nim, specifically from single-pile Nim spanned over two dimensions. Yet, it can itself correspond to a new game of Nim, that being $\mathrm{Nim}_{\mathrm{CHOMP}}$, which is in itself a game of one dimension with multiple piles. This dialectic of dimensions between Nim and Chomp calls for further investigation, as perhaps there exist further simplifications of Chomp that could eventually make it not only more understandable, but also more solvable.

4.4 Determining the Sprague-Grundy values in Slicing Chomp

To determine the Sprague-Grundy values of the different board-slices, one could use the characteristic property in accordance with the Sprague-Grundy theorem. This can be done using backward induction: the pre-terminal state has Sprague-Grundy value 0 and hence, the state containing (0,0) and (0,1) for example has Sprague-Grundy value 1 as that is the minimal excludant, and so on, until all sub-states of the current state have been processed.

This way, the Sprague-Grundy value of the main partial state of the current state can be obtained.

For the other partial states, the Sprague-Grundy value of the pre-terminal state is not zero, as those are not misère games. Instead, the terminal state of such a component game has Sprague-Grundy value 0. The Sprague-Grundy values of those partial states can be determined analogously. Finally, the Sprague-Grundy value of the combined state can be determined using the Nim-sum of the Sprague-Grundy values of all partial states.

5 Approach

In order to investigate how well the first player can play against an adversary that plays perfectly, we have written a program in C++ that performs a number of algorithms. In this thesis, we will analyze the effect of employing the three strategies mentioned before. Strategic play also encapsulates making effective use of the aforementioned winning properties. Moreover, we will implement several methods that prescribe a move to take for the first player and we will implement two methods that designate the optimal move from that state for the second player. The methods that will be implemented for the first player are the following:

(a) **Random:** This method assigns a random object (x, y) as move to perform.

(b) **Monte Carlo:** This method iterates through all available objects (x, y) in the game state. It iteratively performs a move and then performs random playouts between both players. A playout is a play from the current state to the end-state and in random playouts, both player perform random moves. Once the end-state is reached, the score for the move performed before playouts is incremented if the first player won and decremented otherwise. Playouts can be performed a number of times; this is called simulation of plays and in this thesis, we use 100 simulations. After the simulations are performed, a total average score for that move is calculated. The move (x, y) with the highest average score is chosen for the first player.

(c) **Monte Carlo Tree Search:** To describe Monte Carlo Tree Search (MCTS), we need to describe the game in terms of a game tree; that is, a directed game graph whose nodes represent states of the game. The root node of this game tree represents the current state upon which the Monte Carlo Tree Search procedure will be deployed. A state space is the set of states that can be obtained from the current state by performing a move. Monte Carlo Tree Search is a method that is used for games that have vast state spaces. It is designed to avoid exploring every possible move and balances such exploration with exploitation, that is, using well-known good moves. It progressively builds a search tree by performing random simulations to choose the best move. The Monte Carlo Tree Search method can be summarized in four phases that are performed repeatedly per iteration: Selection, Expansion, Simulation and Backpropagation. Below, we describe these four phases [3] [14]:

   i). **Selection:** First of all, Monte Carlo Tree Search starts at the root node and moves down the tree by means of a selection rule until it finds a state that has not been

explored yet. These selection rules aim to control the balance between exploration and exploitation: enough promising moves have to be tried (exploration), yet on the other hand, we wish to proceed with the move that currently leads to the best result (exploitation). Initially for a node, no children have been visited. Hence, expansion, simulation and backpropagation occur directly, to give them an initial score. Based on this, selection can happen imminently. There exist several selection strategies, two of which are described below:

- UCT (Upper Confidence Bound applied to Trees): This is the selection strategy that is most commonly used. Let I be the set of nodes that can be reached directly from the root node r. Then UCT selects a child k such that:

  $$UCT(k) = max_{i \in I}(UCT(i) = v_i + C \cdot \sqrt{\frac{\ln(n_r)}{n_i}}),$$

  where:

  * n_r is the total number of visits to the parent node and the logarithm ensures that exploration decreases as data increases.
  * v_i is the average score of node i and n_i is the number of visits to node i.
  * C is a constant (called the exploration parameter) that can be tuned experimentally, but is typically set to $\sqrt{2}$.

  Note that the exhibited UCT formula looks very similar to the formula of confidence intervals. The formula of a confidence interval CI is as follows:

  $$CI = \text{sample mean} \pm \text{confidence level value} \cdot \sqrt{\frac{\text{variance}}{\text{sample size}}} = \bar{x} \pm z \cdot \sqrt{\frac{s^2}{n}}.$$

  Here, the average reward v_i of a node i corresponds to the sample mean $\bar{x}$, C corresponds to z as both are constants, the logarithm corresponds to the variance and the number of visits to the root node n_r aligns with the sample size n. There is one crucial difference between UCT and the confidence interval formula: the confidence interval formula has a $\pm$-sign to catch values on both sides of the mean, while the UCT formula has only a $+$-sign, meaning that it is one-sided. The reason for this is that UCT looks only at the upper bound of confidence, meaning that it looks only right from the mean. Hence, only a $+$-sign is required.

- KL-UCT (Kullback-Leibler UCT): This selection strategy combines UCT with a probabilistic distance measure, called the Kullback-Leibler divergence. The core difference between UCT and KL-UCT is that the former relies on the confidence interval of rewards, using $\ln(n_r)$ to represent uncertainty, while the latter replaces this confidence interval by and relies solely on the Kullback-Leibler divergence. This metric is a measure of similarity between different distributions and quantifies how a specific probability distribution diverges from a reference distribution, which makes KL-UCT more adaptive in exploring options based on the distribution of rewards [8].

Despite the difference with UCT, KL-UCT also balances exploration and exploitation. Below, we provide a formula for how this is done. Let I once again be the set of nodes that can be reached directly from the root node r. Then KL-UCT selects a child k such that [8]:

$$KL\text{-}UCT(k) = max_{i \in I}(KL\text{-}UCT(i) = max\{b \in [0,1] : d(v_i, b) \leq \frac{\log(f(n_r))}{n_i}\}),$$

with

$$f(n_r) = 1 + n_r \cdot \log^2(n_r),$$

and

$$d(v_i, b) = v_i \log(\tfrac{v_i}{b}) + (1 - v_i) \cdot \log(\tfrac{1-v_i}{1-b}),$$

where:

* n_r is the total number of visits to the parent node.
* b is the upper confidence bound for the current node (child $i \in I$)
* v_i is the average score of node i and n_i is the number of visits to node i.
* d is the Kullback-Leibler divergence (also called the relative entropy).

ii). **Expansion:** Once the selection procedure has reached a leaf node that is not a terminal state, the expansion strategy adds a new leaf node L, which corresponds to the first state that was encountered during traversal and was not already stored.

iii). **Simulation:** From the newly added leaf L, one or more random playouts are simulated. The score is tracked in exactly the same way as done in the Monte Carlo approach. In standard Monte Carlo Tree Search, exactly one playout is simulated at a leaf, but more playouts are possible.

iv). **Backpropagation:** In this phase, the (averaged) score-result of the simulated playout(s) is propagated back up the tree to the root node from leaf L. Subsequently, the statistics, such as total scores and visit counts, are updated for all nodes that were visited during the selection phase (i.e.: the selected performable player moves from the current state).

In this thesis, we chose the number of simulations to be 1000 for Monte Carlo Tree Search.

(d) **Steal-any:** This method performs an arbitrary winning move every time the first player is to make a move and performs a random losing move (the move that was calculated last) if there are no winning moves available from the current state. Such a move is found by randomly selecting a move and then investigating whether this move is winning or not using recursion, repeatedly until a winning move has been found. This is the most powerful method to test against an adversary playing perfectly.

In these methods, we can make the distinction between strategic plays and non-strategic plays. Strategic plays are performed by accounting for the three strategies that we have described. The three strategies are deployed and if applicable, applied. A method that accounts for these strategies is called a *strategic method*. Otherwise, it is just a method. In this thesis, we analyze how much difference it makes for the first player to play strategically compared to

not-strategically. Moreover, we compare the three methods to each other to find out which of the three is most efficient.

5.1 Perfectly playing adversary

As said, in this thesis, we assume perfect play for the second player. In order to know what perfect play entails for the second player, we need to know what optimal moves are from a given game state. First of all, we want moves to be winning. A winning move is a move-option from the current game state that leaves a P-position behind, so that the next player will be in a losing state and will be forced to leave behind an N-position in his turn. Next, we need to know what optimal moves are. We define these moves in detail below.

5.1.1 Optimal moves in Chomp

In the game of Chomp, there are several properties that determine the quality of a move. First of all as said, whether the move is winning or losing. A winning move is always better than a losing move, regardless of whatever other conditions the losing move may satisfy (which the winning move might not). Next, given that a move is winning, the move is the best possible move if it removes as many objects as possible, meaning that the Chomp-bite is as large as possible for a winning move. The reason for this is that the more objects a player removes in a single move, the faster this player plays –and perhaps wins– perfectly. Furthermore, removing more objects leaves the other player with fewer objects and hence, move-options, to choose from, cornering this player into a loss. When a player has no winning moves to perform, because of being in a P-position, then the player should perform a move in which the smallest possible number of objects is removed. Since the objects in Chomp are partially ordered, this move can be determined: the move that is as far as possible right and as far as possible down, where further right has priority over further down. Effectively, this means selecting the available object (x, y) in which both x and y are as large as possible, with precedence for y. We call this specific object the *last valid*. The reason why the player must remove the last valid when there are no winning moves to perform is because it is expected that the game lasts as long as possible by dragging it on, stalling the potential loss and effectively minimizing the regret for this player.

We implemented two methods for finding the optimal move by which perfect play can be performed. The first of these two uses exhaustive search. The second uses a dynamic programming. Below, we explain these methods.

5.1.2 Exhaustive search approach

Exhaustive search is a brute-force approach that is used to find an element with a specific property from a given set. The notion “exhaustive” hints at brute-force, which tries to solve a problem using its definitions in a straightforward manner, while “search” indicates that there is an aim of finding an element with a specific property. The problem-statement we

are given is to find a move that is optimal from a given state. Thus, the exhaustive search approach towards it becomes as follows:

We have a recursive function *ExhaustiveSearch()* which has a state as parameter and returns 1 if the state is winning for the current player and -1 otherwise. The function operates as follows:

(a) Check whether a terminal state has been reached. If so, then the current player has won, because the previous player removed the poisonous object. Hence, value 1 is returned.

(b) If the first check proved false, then a terminal state has not been reached. In that case, iterate through all possible objects in such a way that the corresponding Chomp-bites go from large to small, and do the following:

   i. If the object has already been removed, then skip it.

   ii. Otherwise, Chomp off the given object and perform a recursive call of *ExhaustiveSearch()*. Multiply the return value by -1, as the score obtained in that call would be that of the adversary. Finally, check if the resulting score is greater than 0: if so, then the optimal winning move has been found and the process ends here, returning the score. Otherwise, the next move (object) is checked and if all moves turn out to be losing, -1 is returned.

   Thus, for the score to be obtained, the function would repeat these same steps, until a terminal state is reached. From thereon, the score is propagated backwards by negating its value and storing that negated value as score for the previous player, until the current state is reached. Clearly, this is nothing than backward induction.

   In case no winning move was, found, i.e.: if the overall score is -1, then the last valid object is searched, as to exhaust the opponent and delay his potential victory. If the last valid is $(0, 0)$, then the current player is forced into a loss.

5.1.2.1 Complexity

For exhaustive search, the worst case, when iterating through the move-options from large to small, would be the case in which always the last valid object is the winning object to remove (or the case in which there are no winning objects). Hence, for an $m \times n$-board, an upper bound for the number of states to look at in that worst case would become the following:

$$((m \cdot n) - 1) \cdot ((m \cdot n) - 2) \cdot \ldots \cdot 1 =$$

$$((m \cdot n) - 1)! \in O(m \cdot n!)$$

5.1.3 Dynamic programming approach

Dynamic programming is an approach that optimizes exhaustive search. It does so by storing and remembering solutions of partial problems so that they do not need to be calculated

more than once. It is especially used in situations in which such partial solutions tend to be recomputed multiple times, because the problem has overlapping sub-problems. Such a function that memorizes partial solutions is called a *memory function*, the memorization of partial solutions itself is called *memoization* and the partial problems can be expressed in a recurrence relation. Below, we provide the recurrence relation for optimal moves in Chomp:

Let $S[n]$ be a boolean-array where $S[0]$ denotes the winning value (boolean) of the current state, n is the maximal number of states that can follow from this state and $S[i]$ is the winning value of next state i, where $i > 0$ and $S[n-1]$ is the winning value of the pre-terminal state. For a state i, a valid move $m \in moveset(i)$ can be considered as a function on states. Thus, we may use $m(i)$ to denote the state resulting from performing move m on state i. Let m_i be our optimal move in state i. Then, starting at $S[0]$, the following recurrence relation applies:

$$(\mathrm{S}[i],\, m_i) = \begin{cases} (\text{False},\, (0,0)) : \text{only poison remains (i.e.: } i = n-1) \\ (\text{False, last valid}) : \forall m \in moveset(i) : S[m(i)] \\ (\text{True},\, min\{m \in moveset(i) : \neg S[m(i)]\}) : \exists m \in moveset(i) :\ \neg S[m(i)] \end{cases} \tag{2}$$

This recurrence relation explains whether or not the current state 0 and thereby, the optimal move m_0, is winning. Hence, by means of this relation, we can determine whether or not an arbitrary state i is winning or not.

5.1.3.1 Top-Down approach

Since the problem of determining the optimal move can be broken down recursively by using its partial solutions and these partial solutions overlap, they can easily be memoized. This is often done in an array or a hash-table. A hash-table is a data structure that implements a dictionary that maps keys to values. In this thesis, we made use of such a table, as it has a complexity of O(1) in the average case, while a vector-search has a complexity of O(N) on average and in the best and worst case they are equal to each other in complexity. Whenever a partial problem needs to be solved, the algorithm checks in the hash-table whether the problem was already solved. In this case that is whether it has already been determined if a state is winning or losing for the player to move from it. If so, then this information can be used directly.

The Top-Down approach is known for various things; it uses recursion in an optimized way as well as an array or hash-table and it does not solve all partial problems, but only those that are needed for the original task. Once the smallest partial problem has been solved in this sequence of solving problems, it propagates the result backward, which happens repeatedly until the solution for the original problem is known. This effectively entails solving the problem, again, by means of backward induction.

5.1.3.2 The dynamic programming algorithm

The algorithm uses a local recursive function *positionIsWinning()* that checks whether or not a given state is winning or losing and a function *generateState()*, that generates the key for the hash-table elements. The hash-table is an *unordered_map* and consists of key-value pairs (string, bool), where the string is a state-encoding and the boolean represents whether the state is winning (True) or losing (False).

The function generateState() counts the number of remaining objects in a row and appends that to the string as a byte. A byte can store up to 255 row-objects which is clearly sufficient for general Chomp-boards and is much smaller than an integer, which is 4 bytes. This is done for every row, to obtain a byte-wise string encoding of the state. Since a Chomp-board is equivalent to its transpose, we also count the number of objects in each column and append that to another string as encoding. The smallest encoding of the two is then passed as the final encoding. Storing states in this way means that if the transpose of a state is actually reachable in the future, then it is already stored and does not need to be reprocessed. This minimizes the amount of memory and saves calculations.

The function positionIsWinning() starts by generating an encoding of the given state it is in and checks whether or not this state is already in the hash table. It does this by looking it up using the state as unique key and obtaining its value True or False. If it cannot find the state, then it has not been processed before and still needs to be processed. In that case, the algorithm iterates through all possible objects in such a way that the corresponding Chomp-bites go from large to small and performs a bite, then checks whether or not this bite leads to a winning or losing state. This is done by recursively calling positionIsWinning(). If a bite leads to a losing state (for the adversary), then the state is stored in the hash table as a winning one and otherwise, this process is continued for the next move. If no moves remain and the state is not winning, then it is marked as a losing state and inserted as such in the hash table. Effectively, positionIsWinning() implements the recurrence relation described above.

The main function of dynamic programming operates much like the function positionIsWinning(). It iterates through all possible objects so that the corresponding Chomp-bites go from large to small, performs a move and then checks whether the obtained state is winning for the adversary. If in this procedure a winning move is found, then that move is returned as optimal move. Else, the last valid object is returned as move, because it delays a potential victory of the adversary.

5.2 Performing competitions

We implemented ten competitions between the first player and the adversary. The first four compare the random, Monte Carlo, Monte Carlo Tree Search (UCT and KL-UCT) for the first player cleverly or non-cleverly against exhaustive search, and the second four do the same,

except that the four methods are compared to dynamic programming. The ninth competition compares the steal-any approach against dynamic programming and the tenth competition compares the strategy stealing approach, which is implemented as dynamic programming competing against dynamic programming. In the steal-any approach and the strategy stealing approach, both players use the hash table as a common table to minimize the use of memory.

5.2.1 Measurement of performance

We measure performance by means of a win-loss ratio for the first player and efficiency by the following two metrics, ordered by priority:

(a) In case the first player won: the number of moves it took to win; this should be as low as possible. A low number of moves would mean that the first player won with relative ease.

In case the first player lost: how many moves it took the adversary to win: this should be as high as possible, as a high number of moves would indicate that the adversary struggled in order to win and that the first player was delaying his loss effectively.

(b) The time it took for the methods to be performed: a method is most efficient if its time consumption is as low as possible. A lot of time consumption indicates that methods do much work, which is undesired: we want a method to be as efficient as possible with the least amount of effort.

To account for this, we measure the time for player 1, which indicates how long it took a (strategic) method to account for all its moves in a playout and we measure the time for player 2, which indicates how long it took for exhaustive search or dynamic programming to do the same. The number of moves it took the winning player is also tracked. The performance of the first player can then be measured by means of the following two metrics, where the time is measured in seconds and *win* is a boolean that indicates that the first player won:

$$P_1(!win) = \frac{Time(P_2)}{Time(P_1)} \cdot moves(P_2),$$

$$P_1(win) = \frac{Time(P_2)}{Time(P_1) \cdot moves(P_1)}$$

For the first player, we want the scores to be as high as possible, as this would either mean that the adversary struggled if the first player lost, or that the first player won in very few moves, if the first player won. Aside from this, it would also indicate that the time for the adversary was large compared to that of the first player, which also indicates that the adversary struggled.

6 Experiments

For the experiments, we made use of the metric of performance for player 1. We first compare the methods and strategic methods Random, Monte Carlo and Monte Carlo Tree Search (with UCT and KL-UCT) against each other, and then, we compare strategy stealing to the steal-any approach. We use 100 simulations for Random. If the first player won in more than 65 of those simulations, then we consider this player to be the overall winner of Random-comparison. For strategic methods, square games are not compared, as these are always first-player wins by strategy 2 and hence, trivial. For the experiments, we decided to halt simulations if their execution times exceeded two hours. Hereunder, we provide the required experiments, with scores of first-player wins being highlighted in yellow. Hence, using scores, we can compare the methods directly to each other.

6.1 Experiment 1: Comparison of methods

Below, we provide the results of comparison of methods. Infeasible calculations are left out.

Board	Score
1x1	0
1x2	1.0187
1x3	0.793
1x4	0.350
1x5	0.280
1x6	0.310
1x7	0.252
1x8	0.213
1x9	0.182
1x10	0.211
1x11	0.207
1x12	0.160
1x13	0.156
1x14	0.155
1x15	0.139
2x1	1.045
2x2	1.418
2x3	1.019
2x4	7.421
2x5	7.688
2x6	23.149
2x7	24.135
2x8	79.509
2x9	81.317
2x10	333.782
2x11	240.894
2x12	1133.26
2x13	1181.89
2x14	4884.41
2x15	5886.36

3x1	1.049
3x2	1.477
3x3	3.086
3x4	40.680
3x5	47.609
3x6	583.231
3x7	853.996
3x8	15203.637
3x9	22285.320
3x10	210945.594
3x11	281537
3x12	6.024e+06
3x13	5677215.5
3x14	6.825e+07
3x15	2.099e+08
4x1	0.421
4x2	14.693
4x3	54.058
4x4	264.508
4x5	4121.81
4x6	8801.97
4x7	250649
4x8	345568
4x9	1.032e+07
5x1	0.64215
5x2	15.5276
5x3	218.151
5x4	6485.54
5x5	8406.66
5x6	455262
5x7	481784
5x8	7.698e+07

6x1	0.480
6x2	157.65
6x3	2911.77
6x4	23625.3
6x5	1.397e+06
6x6	3.502e+06
7x1	0.443
7x2	124.4
7x3	360.329
7x4	676658
7x5	787336
8x1	0.452
8x2	721.057
8x3	92462.7
8x4	8.936e+06
8x5	2.208e+08
9x1	0.378
9x2	1060.69
9x3	107426
9x4	1.878e+08
10x1	0.404
10x2	4388.52
10x3	1.917e+06
11x1	0.372
11x2	3668.42
11x3	4.168e+06
12x1	0.373
12x2	11576.7
12x3	5.743e+07

13x1	0.406
13x2	13311.1
13x3	5.839e+07
14x1	0.412
14x2	53909.3
15x1	0.420
15x2	78150.8

Board	Score
1x1	0
1x2	0.015
1x3	0.006
1x4	0.002
1x5	0.001
1x6	0.001
1x7	0.001
1x8	0.001
1x9	0.001
1x10	0.001
1x11	0
1x12	0
1x13	0
1x14	0
1x15	0
2x1	0.017
2x2	0.007
2x3	0.012
2x4	0.008
2x5	0.005
2x6	0.018
2x7	0.004
2x8	0.003
2x9	0.002
2x10	0.002
2x11	0.002
2x12	0.001
2x13	1204.640
2x14	690.383
2x15	0.013

3x1	0.007
3x2	0.006
3x3	0.003
3x4	0.012
3x5	0.012
3x6	0.062
3x7	0.012
3x8	0.329
3x9	2322.426
3x10	91.564
3x11	0
3x12	42.556
3x13	66.779
4x1	0.004
4x2	0.013
4x3	0.046
4x4	0.089
4x5	7.297
4x6	7.007
4x7	928.580
4x8	2102.523
4x9	13.446
4x10	580.827
4x11	934599.625
4x13	1.709
4x15	0

5x1	0.002
5x2	0.009
5x3	0
5x4	1.396
5x5	10.619
5x6	0.173
5x7	421.280
5x8	0.844
5x9	371.208
5x10	207.918
5x11	82977.617
6x1	0.002
6x2	0.007
6x3	0.015
6x4	0.348
6x5	12.931
6x6	12095.133
6x7	205.209
6x8	0.221
6x9	6.204
7x1	0.001
7x2	0.005
7x3	5.759
7x4	942.833
7x5	19.674
7x6	199981.734
7x7	16.488
7x8	0
7x9	0.192
7x10	47.187

8x1	0.001
8x2	0.004
8x3	0
8x4	3442.239
8x5	80680.953
8x6	0.102
8x7	0
8x9	739.258
8x10	0
9x1	0.001
9x2	0.085
9x3	1813.291
9x4	17.066
9x5	11.550
9x6	0
10x1	0.001
10x2	0
10x3	488.585
11x1	0.001
11x2	0.030
11x3	10854.299
11x4	1690.655
12x1	0.009
12x2	0.021
12x3	0.004
12x5	2.179
12x7	47782.578

13x1	0.001
13x2	0.001
13x3	1.095
13x4	0.612
13x6	5347.368
13x8	0
14x1	0
14x2	0.001
14x3	239758.500
15x1	0
15x2	0.001
15x5	121340.352

Table 1: Results of Random against Exhaustive Search (left) and Monte Carlo against Exhaustive Search (right)

Board	Score
1x1	0
1x2	0
1x3	0
1x4	0
1x5	0
1x6	0
1x7	0
1x8	0
1x9	0
1x10	0
1x11	0
1x12	0
1x13	0
1x14	0
1x15	0
2x1	0
2x2	0
2x3	0
2x4	0
2x5	0
2x6	0
2x7	0
2x8	0.013
2x9	0
2x10	0.052
2x11	0.028
2x12	1.033
2x13	0
2x14	81.022
2x15	0.019
3x1	0
3x2	0
3x3	0
3x4	0
3x5	0.014
3x6	0
3x7	0
3x8	0.057
3x9	0
3x10	0.010
3x11	22.779
3x12	0
3x13	0.094
3x14	33.447
4x1	0
4x2	0
4x3	0
4x4	0
4x5	0.089
4x6	0.072
4x7	0.673
4x8	0.957
4x9	0

5x1	0
5x2	0
5x3	0
5x4	0.002
5x5	0.102
5x6	7.723
5x7	8.359
5x8	35.115
5x9	21.503
6x1	0
6x2	0.009
6x3	0.008
6x4	0.011
6x5	0.025
6x6	0.046
6x7	0.003
6x9	2.921
6x10	0.001
7x1	0
7x2	0.001
7x3	1.559
7x4	0
7x5	0.084
7x8	0.045
7x10	71.837
8x1	0
8x2	0.001
8x3	2.217
8x4	0.357
8x5	0.077
8x7	13.631
9x1	0
9x2	0.011
9x3	2.328
9x4	359.764
9x5	0.045
9x7	56.209
10x1	0
10x2	0.002
10x3	152.333
10x4	65.633
11x1	0
11x2	0.029
11x3	24.305
11x4	37.167
12x1	0
12x2	0.034
12x3	0.001

13x1	0
13x2	0.001
13x3	0
13x4	0
13x5	0.024
13x6	57.087
14x1	0
14x2	0
14x3	6321.687
14x4	11.216
14x5	0
15x1	0
15x2	570.073
15x3	0.556
15x5	140.516

Board	Score
1x1	0
1x2	0
1x3	0
1x4	0
1x5	0
1x6	0
1x7	0
1x8	0
1x9	0
1x10	0
1x11	0
1x12	0
1x13	0
1x14	0
1x15	0
2x1	0
2x2	0
2x3	0
2x4	0
2x5	0
2x6	0
2x7	0
2x8	0
2x9	0
2x10	0
2x11	0
2x12	0.003
2x13	21.399
2x14	0
2x15	0.005
3x1	0
3x2	0
3x3	0
3x4	0
3x5	0.010
3x6	0.065
3x7	0.010
3x8	0.030
3x9	0.008
3x10	0.789
3x11	0.364
3x12	0
3x13	0.019
3x14	5339.434
3x15	1432.151
4x1	0
4x2	0
4x3	0
4x4	0
4x5	0.033
4x6	0.040
4x7	38.812
4x8	24.037
4x9	0.151
4x11	0
4x13	0

5x1	0
5x2	0
5x3	0
5x4	0.020
5x5	0.031
5x6	11.077
5x7	10.272
5x9	0
5x10	5.286
5x11	0
6x1	0
6x2	0.001
6x3	0.039
6x4	0.160
6x5	0.018
6x6	0.002
6x7	0.004
6x11	0.002
7x1	0
7x2	0
7x3	0.102
7x4	4.054
7x5	18.775
7x6	17.971
8x1	0
8x2	0
8x3	0
8x4	0.088
8x5	0.043
8x10	126.544

9x1	0
9x2	0.002
9x3	0.001
9x4	0
9x7	0.673
10x1	0
10x2	0
10x3	0.587
11x1	0
11x2	3.978
11x3	38.711
11x4	2154.12
12x1	0
12x2	0
12x3	0.026
12x5	407.616
13x1	0
13x2	0
13x3	10055.299
13x4	0
14x1	0
14x2	0.861
15x1	0
15x2	4.737
15x3	2.427
15x4	0

Table 2: Results of Monte Carlo Tree Search against Exhaustive Search (left: UCT, right: KL-UCT)

Board	Random	MC
1x1	0	0
1x2	0.427	0.004
1x3	0.413	0.002
1x4	0.440	0.001
1x5	0.350	0.001
1x6	0.316	0.001
1x7	0.302	0.001
1x8	0.270	0
1x9	0.250	0.001
1x10	0.248	0
1x11	0.237	0
1x12	0.204	0
1x13	0.182	0
1x14	0.200	0
1x15	0.189	0
2x1	0.427	0
2x2	1.043	0.017
2x3	0.788	0.029
2x4	2.124	0.018
2x5	1.630	0.043
2x6	2.785	0.044
2x7	2.670	0.017
2x8	3.249	0.004
2x9	3.153	0.004
2x10	3.848	0.003
2x11	3.674	0.048
2x12	4.813	0.002
2x13	3.378	0.012
2x14	5.510	0.040
2x15	6.447	0.001
3x1	0.431	0.003
3x2	1.042	0.043
3x3	1.237	0.030
3x4	4.165	0.019
3x5	2.823	0.034
3x6	6.542	0.058
3x7	3.968	0.057
3x8	2.975	0.005
3x9	13.212	0.069
3x10	14.284	0.007
3x11	11.010	0.042
3x12	12.090	0.020
3x13	13.236	0.269
3x14	17.701	0.277
3x15	14.038	0.002
4x1	0.924	0.002
4x2	3.248	0.015
4x3	4.510	0.018
4x4	3.531	0.067
4x5	10.483	0.024
4x6	5.259	0.041
4x7	17.337	0.134
4x8	13.344	0.010
4x9	23.347	0.292
4x10	70.185	0.330
4x11	25.632	0.420
4x12	47.766	0.408
4x13	81.877	0.633
4x14	305.109	1.698
4x15	359.128	0.001
5x1	0.348	0.001
5x2	2.683	0.032
5x3	4.199	0.001
5x4	9.732	0.021
5x5	7.263	0.018
5x6	16.996	0.063
5x7	10.616	0.067
5x8	22.572	0.099
5x9	34.339	0.009
5x10	82.229	0.135
5x11	53.800	1.331
5x12	280.508	0.002
5x13	255.875	0.299
5x14	606.646	0.010
5x15	363.899	0.608
6x1	0.336	0
6x2	4.513	0.009
6x3	6.921	0.051
6x4	9.989	0.012
6x5	16.560	0.010
6x6	12.467	0.274
6x7	129.580	0.006
6x8	199.312	0.024
6x9	169.493	0.214
6x10	43.755	0.037
6x11	267.228	1.350
6x12	1245.208	0.004
6x13	328.491	0.371
6x14	1394.620	0.130
6x15	3727.438	0.008
7x1	0.305	0
7x2	4.090	0.048
7x3	7.856	0.003
7x4	20.123	0.020
7x5	10.912	0.058
7x6	68.632	0.011
7x7	40.084	0.337
7x8	155.790	0.566
7x9	478.332	0.826
7x10	199.307	1.056
7x11	2003.461	0.004
7x12	283.753	1.508
7x13	4065.635	5.736
7x14	2993.662	8.329
7x15	7741.299	0
8x1	0.253	0
8x2	5.848	0.094
8x3	34.119	0.062
8x4	65.511	0.036
8x5	55.697	0.059
8x6	19.393	0.472
8x7	5.437	0.015
8x8	118.988	0.620
8x9	902.639	0
8x10	1934.556	2.425
8x11	1801.062	12.591
8x12	5249.177	14.728
8x13	12543.563	27.243
8x14	48.228	47.663
8x15	247630.328	88.375
9x1	0.377	0
9x2	5.691	0.014
9x3	15.945	0.078
9x4	273.551	0.223
9x5	60.598	0.014
9x6	93.309	0.080
9x7	179.223	0.036
9x8	933.552	0.048
9x9	178.387	2.656
9x10	2800.922	7.108
9x11	3750.738	0.019
9x12	2135.686	7.008
9x13	9969.188	23.095
9x14	35364.129	0.186
9x15	68260.422	100.649
10x1	0.314	0
10x2	6.748	0.004
10x3	21.888	11.339
10x4	46.720	0.114
10x5	34.573	0.026
10x6	564.954	0.009
10x7	34.074	0.038
10x8	1587.670	4.069
10x9	2322.248	0.937
10x10	1344.245	6.604
10x11	7130.208	4.571
10x12	413.395	0.139
10x13	39793.820	92.634
10x14	15863.872	207.624
10x15	96495.477	0.783
11x1	0.230	0
11x2	5.974	0.026
11x3	18.534	0.005
11x4	1.346	0.115
11x5	281.237	0.502
11x6	712.565	1.154
11x7	1377.26	2.128
11x8	4934.688	0.177
11x9	7119.283	23.890
11x10	1089.082	2.514
11x11	3026.138	16.994
11x12	69446.133	0.105
11x13	123094.484	259.624
11x14	194183.766	684.800
11x15	522852.000	126.209
12x1	0.267	0
12x2	10.952	0.012
12x3	8286.979	43.770
12x4	96.242	0.789
12x5	417.868	2.011
12x6	348.185	4.237
12x7	1042.259	1.759
12x8	303.690	1.415
12x9	1183.146	9.062
12x10	29728.244	0.258
12x11	48514.316	52.038
12x12	54722.730	0.002
12x13	281481.062	0.392
12x14	59781.391	0.002
12x15	1502754.625	2732.204
13x1	0.263	0
13x2	11.481	0.049
13x3	26.902	0.004
13x4	283.444	0.003
13x5	6.862	0.026
13x6	1185.731	2.793
13x7	2287.520	0.514
13x8	8552.695	0.058
13x9	11697.570	13.097
13x10	58646.098	4.232
13x11	19333.699	15.205
13x12	270914.406	236.455
13x13	141717.047	90.850
13x14	129021.414	176.485
13x15	234103.359	202.357
14x1	0.257	0
14x2	9.029	0.016
14x3	32.110	0.140
14x4	607.284	0.567
14x5	249.999	0.431
14x6	3629.038	0.459
14x7	7524.585	12.168
14x8	19756.936	0
14x9	33521.918	0.851
14x10	170916.312	4.382
14x11	139717.375	0.057
14x12	804654.375	189.870
14x13	1120967	0.023
14x14	161853.719	815.385
14x15	2023456.875	3608.762
15x1	0.236	0
15x2	15.053	0.002
15x3	74.205	0.144
15x4	668.623	0.468
15x5	1217.583	0.705
15x6	4934.113	0.529
15x7	1328.695	8.988
15x8	38247.645	44.443
15x9	10696.299	0.094
15x10	158013.281	2.899
15x11	535056.938	0.005
15x12	1356948.5	0.150
15x13	323708.688	4110.125
15x14	5543465.500	2131.353
15x15	743088.812	12416.914

Table 3: Scores of Random and Monte Carlo against Dynamic Programming

Board	UCT	KL-UCT
1x1	0	0
1x2	0	0
1x3	0.001	0
1x4	0	0
1x5	0	0
1x6	0	0
1x7	0	0
1x8	0	0
1x9	0	0
1x10	0	0
1x11	0	0
1x12	0	0
1x13	0	0
1x14	0	0
1x15	0	0
2x1	0	0
2x2	0	0
2x3	0	0
2x4	0	0
2x5	0	0
2x6	0	0
2x7	0	0
2x8	0	0
2x9	0.001	0
2x10	0.001	0
2x11	0	0.001
2x12	0.001	0
2x13	0.001	0.001
2x14	0.002	0.001
2x15	0.001	0.001
3x1	0	0
3x2	0	0
3x3	0	0
3x4	0	0
3x5	0	0
3x6	0.001	0.001
3x7	0	0
3x8	0.001	0.002
3x9	0.002	0.001
3x10	0	0.001
3x11	0.001	0.001
3x12	0	0.001
3x13	0.002	0.003
3x14	0.004	0.002
3x15	0	0.007
4x1	0	0
4x2	0	0
4x3	0	0
4x4	0	0
4x5	0.001	0
4x6	0.001	0.001
4x7	0.002	0
4x8	0	0
4x9	0	0.003
4x10	0.006	0.001
4x11	0.001	0
4x12	0.007	0.008
4x13	0.018	0
4x14	0	0.009
4x15	0.003	0.003
5x1	0	0
5x2	0	0
5x3	0	0
5x4	0.001	0.001
5x5	0.001	0.001
5x6	0	0
5x7	0.001	0
5x8	0.001	0
5x9	0.003	0.004
5x10	0.006	0.009
5x11	0.001	0.002
5x12	0.032	0.001
5x13	0.005	0.006
5x14	0.019	0.025
5x15	0.044	0.033
6x1	0	0
6x2	0	0
6x3	0	0
6x4	0.001	0
6x5	0.001	0.001
6x6	0.001	0.003
6x7	0	0
6x8	0.008	0.008
6x9	0.010	0.001
6x10	0.034	0.016
6x11	0	0.045
6x12	0.065	0.022
6x13	0	0.096
6x14	0.124	0.127
6x15	0.084	0.004
7x1	0	0
7x2	0	0
7x3	0	0
7x4	0	0.001
7x5	0	0.001
7x6	0	0.006
7x7	0	0.006
7x8	0.001	0.016
7x9	0.006	0.024
7x10	0.011	0.019
7x11	0.073	0.125
7x12	0.003	0.063
7x13	0.069	0.146
7x14	0.002	0.485
7x15	0.219	0.575
8x1	0	0
8x2	0	0
8x3	0.002	0
8x4	0.002	0.002
8x5	0.006	0.004
8x6	0	0.002
8x7	0	0.010
8x8	0.010	0.014
8x9	0	0
8x10	0	0.029
8x11	0.246	0.054
8x12	0.108	0.122
8x13	0.228	0.401
8x14	0.147	0.957
8x15	0.948	1.926
9x1	0	0
9x2	0	0.001
9x3	0	0
9x4	0.003	0.007
9x5	0.012	0.022
9x6	0.014	0.004
9x7	0.010	0.007
9x8	0.005	0.017
9x9	0	0.073
9x10	0.005	0.108
9x11	0.001	0.238
9x12	0.299	0.652
9x13	0.701	0.770
9x14	0.727	1.266
9x15	0	0.131
10x1	0	0
10x2	0.001	0
10x3	0.002	0.002
10x4	0.005	0.007
10x5	0.005	0.008
10x6	0.012	0.026
10x7	0.002	0.036
10x8	0.036	0.041
10x9	0.082	0.139
10x10	0.003	0.157
10x11	0.031	0.156
10x12	0.057	0.036
10x13	0.584	0.058
10x14	0.099	1.520
10x15	0.463	4.219
11x1	0	0
11x2	0.001	0.001
11x3	0.001	0.002
11x4	0.0010	0.005
11x5	0.004	0.029
11x6	0.013	0.036
11x7	0.065	0.128
11x8	0.124	0.072
11x9	0.157	0.253
11x10	0.295	0.373
11x11	0.061	0.370
11x12	0.074	1.001
11x13	1.408	4.071
11x14	0.099	6.347
11x15	1.875	26.164
12x1	0	0
12x2	0	0
12x3	0.001	0.002
12x4	0.003	0.007
12x5	0.010	0.009
12x6	0.069	0.121
12x7	0.003	0.084
12x8	0.011	0.113
12x9	0.766	1.003
12x10	0.138	1.413
12x11	1.532	1.952
12x12	3.139	3.758
12x13	6.155	6.202
12x14	18.728	34.772
12x15	3.821	9.202
13x1	0	0
13x2	0	0
13x3	0.004	0.002
13x4	0.003	0.011
13x5	0.026	0.041
13x6	0.060	0.062
13x7	0.073	0.079
13x8	0.303	0.417
13x9	0.042	0.375
13x10	2.758	2.776
13x11	0.058	1.031
13x12	4.326	6.795
13x13	8.925	11.035
13x14	10.785	16.212
13x15	2.877	32.595
14x1	0	0
14x2	0.001	0.004
14x3	0.002	0.016
14x4	0	0.002
14x5	0.018	0.006
14x6	0	0.020
14x7	0.061	0.022
14x8	0.026	1.171
14x9	0.154	0.710
14x10	0.549	2.095
14x11	0.444	9.163
14x12	4.425	41.081
14x13	13.185	33.435
14x14	1.077	1.580
14x15	0.179	53.277
15x1	0	0
15x2	0.001	0.001
15x3	0.005	0.007
15x4	0	0.009
15x5	0	0.037
15x6	0.037	0.297
15x7	0.504	0.615
15x8	0.273	0.673
15x9	0.272	2.989
15x10	0.960	4.797
15x11	3.686	7.659
15x12	2.056	2.968
15x13	1.835	31.553
15x14	0.037	45.459
15x15	41.440	73.782

Table 4: Scores of Monte Carlo Tree Search against Dynamic Programming per UCT and KL-UCT

6.1.1 Comparison for the second player

In the exhaustive search runs, many larger board-dimensions turned out to be infeasible. We observe that overall, for larger board-dimensions, methods tend to achieve much higher scores against exhaustive search than against dynamic programming. Since the methods themselves are not different, this indicates that it took exhaustive search more time to search for a move to perform than it took dynamic programming. Hence, we can draw out a general inference about dynamic programming in contrast to exhaustive search:

(a) Dynamic programming allows for transcending to larger board-dimensions in analysis, meaning that larger board-dimensions become feasible. This occurs due to the hash-table used, which allows for earlier computed values to be reused without the need of recomputation. This also means that Dynamic programming finds the move to perform much faster than exhaustive search does.

We conclude that dynamic programming is a far more efficient method than exhaustive search, vigorously outperforming it in computational efficiency and time-wise performance.

6.1.2 Comparison for the first player

Due to dynamic programming strongly outperforming exhaustive search, we compare methods only in the situation where the adversary plays using optimal dynamic programming moves. From the above tables, we observed that for Monte Carlo Tree Search, KL-UCT overall tended to obtain higher scores than UCT. In relatively few cases, UCT obtained a higher score than KL-UCT, however, due to the low frequency of this occurring, we believe that these cases were mostly coincidental rather than causal. KL-UCT and also found a few more winning moves for the first player than UCT. Hence, KL-UCT exhibited superior performance. Below, we provide a table of an comparison of methods between each other, for larger board-sizes, in which we use only KL-UCT in Monte Carlo Tree search for comparing.

Board	R-MC	R-MCTS	MC-R	MC-MCTS	MCTS-R	MCTS-MC
10x10	47067.7	2.776e+06	1.338e-05	7.08903	1.887e-07	0.043
10x11	55275.5	2.832e+06	1.371e-05	14.488	5.118e-06	0.001
10x12	59935.8	3.440e+06	1.439e-05	18.777	1.173e-07	0.099
10x13	51956.2	3.746e+06	1.104e-05	1.782	1.225e-07	0.008
10x14	65860.5	5.765e+06	9.959e-06	9.122	1.048e-07	0.015
10x15	70882.8	6.638e+06	8.469e-06	3.642	9.955e-08	0.044
11x11	51912.8	3.566e+06	1.192e-05	20.086	1.271e-07	0.051
11x12	53825.2	5.297e+06	1.302e-05	5.422	1.223e-06	0.063
11x13	81663.2	4.681e+06	1.085e-05	25.222	2.075e-06	0.028
11x14	88370	7.522e+06	7.106e-06	5.917	1.038e-07	0.002
11x15	82192.3	7.825e+06	8.75194e-06	9.247	6.308e-07	0.045
12x12	82793.6	1.591e+06	1.487e-05	9.904	8.484e-08	0.048
12x13	69423.1	5.545e+06	1.135e-05	10.052	8.995e-08	0.057
12x14	82457.7	8.642e+06	9.216e-06	5.969	7.743e-08	0.029
12x15	90759.1	9.287e+06	1.033e-05	4.543	9.452e-08	0.031
13x13	90391.4	6.374e+06	1.107e-05	9.974	1.153e-07	0.022
13x14	160749	7.802e+06	6.923e-06	5.122	9.163e-08	0.062
13x15	150264	8.528e+06	7.536e-06	6.485	1.112e-07	0.021
14x14	150148	9.584e+06	8.512e-06	9.291	6.63094e-08	0.046
14x15	192101	9.468e+06	6.399e-06	5.609	6.949e-08	0.002
15x15	63935.6	1.185e+07	7.446e-06	7.023	7.380e-08	0.123

Table 5: Scores of methods compared against each other: R (Random), MC (Monte Carlo) and MCTS (Monte Carlo Tree Search with KL-UCT)

From the tables above, we observe several things. First of all, we can make several inferences about the performance of different methods for the first player. For the first player, it is clear

that the Random method always lost against Monte Carlo and Monte Carlo Tree Search. Monte Carlo always won against Random and against Monte Carlo Tree Search. And Monte Carlo Tree Search almost always won against Random, except for a few cases. Monte Carlo Tree Search always lost against Monte Carlo. Essentially, this means that the Monte Carlo method performed the best from the three methods, with the second best being Monte Carlo Tree Search and the worst of them being the Random method.

Now, we can make inferences about the comparison against an optimal player. We observe that overall, Random had the highest scores. However, since Random is clearly the worst method from the three, we believe that these high scores are caused by its time-wise efficiency: Random is very fast compared to the other methods and hence, obtains higher scores. We observed that the second-highest scores were obtained by the Monte Carlo method, which was also the best at forcing wins, especially for smaller board-dimensions. The lowest scores were obtained by Monte Carlo Tree Search. From this, we believe that the Monte Carlo method performed best, as it already beat all other methods in the comparison of methods between each other and also beat Monte Carlo Tree Search in the comparison against the optimal adversary.

6.1.3 Sub-experiment: Performance on square-boards

For the performance on square boards, we plotted the results in a graph, comparing the performances of all prescribed methods against dynamic programming. In this graph, a surrounding square indicates that the first player won for the corresponding board-dimension.

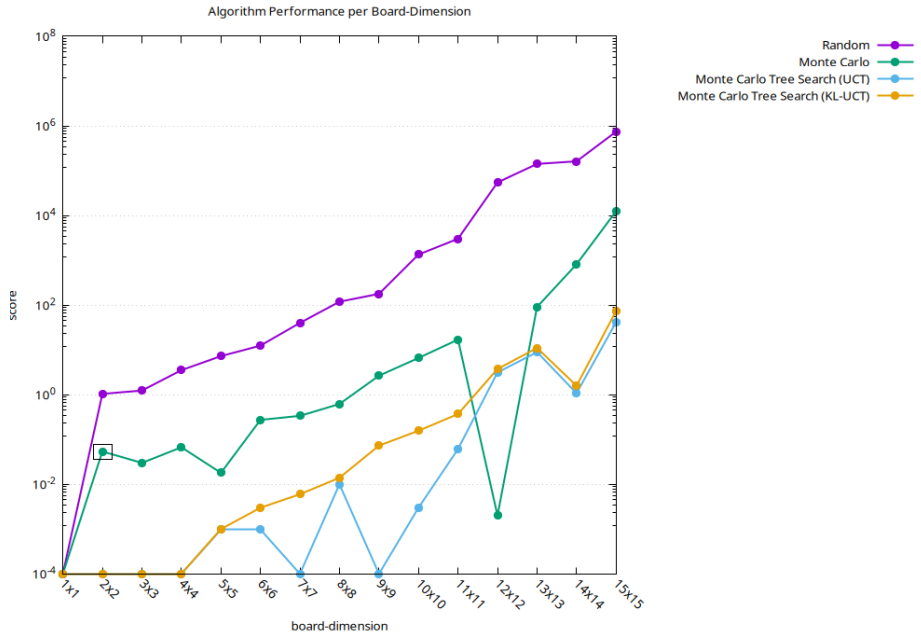

Figure 7: Performance on square boards for various methods against dynamic programming

6.2 Experiment 2: Comparison of strategic methods

In this experiment, Random, Monte Carlo and Monte Carlo Tree Search with KL-UCT are compared against each other cleverly. Below, we provide the obtained results.

Board	Random	MC	MCTS
1x2	0.253	0.088	0.095
1x3	0.301	0.116	0.087
1x4	0.254	0.114	0.111
1x5	0.281	0.102	0.108
1x6	0.105	0.118	0.121
1x7	0.290	0.107	0.131
1x8	0.282	0.092	0.127
1x9	0.241	0.182	0.832
1x10	0.220	0.122	0.817
1x11	0.233	0.116	0.703
1x12	0.269	0.097	0.893
1x13	0.268	0.129	0.935
1x14	0.266	0.125	0.910
1x15	0.257	0.096	0.930
2x1	0.233	0.071	0.74
2x3	0.601	1.080	1.920
2x4	0.472	1.272	1.723
2x5	0.593	1.863	1.597
2x6	0.739	1.789	1.912
2x7	0.650	2.221	2.316
2x8	0.788	1.826	3.499
2x9	0.736	2.455	2.536
2x10	0.806	3.136	4.445
2x11	0.885	2.811	2.738
2x12	0.930	3.410	3.960
2x13	0.985	3.410	4.293
2x14	0.983	3.733	3.884
2x15	1.105	3.286	5.021
3x1	0.115	0.065	0.084
3x2	0.457	0.931	1.107
3x4	0.547	0.023	0
3x5	1.365	0.004	0.001
3x6	5.930	0.028	0.001
3x7	3.354	0.004	0.001
3x8	6.707	0.008	0
3x9	3.342	0.005	0.001
3x10	9.307	0.109	0.001
3x11	6.441	0.136	0.002
3x12	10.555	0.043	0.003
3x13	8.263	0.118	0.003
3x14	10.125	0.002	0.003
3x15	10.094	0.001	0.001
4x1	0.140	0.062	0.093
4x2	0.483	1.381	1.845
4x3	0.562	0.001	0.001
4x5	0.709	0.020	0.002
4x6	3.137	0.009	0.001
4x7	0.843	0.033	0.002
4x8	8.812	0.025	0.002
4x9	12.536	0.267	0
4x10	1.213	0.007	0.006
4x11	20.602	0	0.001
4x12	25.630	0	0.007
4x13	1.504	0.003	0
4x14	1.737	0.225	0
4x15	53.198	0.013	0.023
5x1	0.122	0.078	0.089
5x2	0.377	1.102	1.440
5x3	2.327	0.022	0.001
5x4	0.660	0.017	0
5x6	0.797	0.056	0
5x7	6.966	0.123	0
5x8	17.474	0.044	0.001
5x9	17.102	0.282	0.006
5x10	27.551 7.993	0.548	0
5x11	28.454	0.057	0
5x12	65.982	0.013	0.021
5x13	87.645	2.168	0.021
5x14	101.116	0.029	0.024
5x15	80.604	2.016	0
6x1	0.146	0.073	0.081
6x2	0.480	57.275	56.871
6x3	3.545	0.001	0
6x4	1.013	0.009	0
6x5	9.213	0.259	0
6x7	9.669	0.165	0
6x8	22.340	0.030	0.006
6x9	25.149	0.168	0.011
6x10	78.360	0.005	0.021
6x11	135.779	2.312	0.007
6x12	212.206	4.372	0.032
6x13	486.311	0.045	0
6x14	1519.648	0.290	0
6x15	1768.529	2.157	0
7x1	0.139	0.074	0.108
7x2	0.492	82.150	85.450
7x3	4.301	0.011	0.001
7x4	9.310	0.131	0.001
7x5	5.812	0.155	0.002
7x6	24.398	0.038	0.006
7x8	36.861	0.076	0.003
7x9	68.567	0.384	0.014
7x10	51.058	0.792	0.016
7x11	348.696	6.008	0
7x12	1074.021	1.110	0.010
7x13	2267.602	5.168	0
7x14	3579.417	6.762	0.188
7x15	1343.580	4.866	0.075
8x1	0.143	0.083	0.097
8x2	0.498	2.011	3.020
8x3	7.728	0.127	0
8x4	10.409 2	0.007	0.002
8x5	15.354	0.551	0.004
8x6	31.764	0.129	0.003
8x7	27.063	0.031	0.019
8x9	172.783	0.009	0.007
8x10	1060.496	3.578	0.113
8x11	1062.143	7.653	0.118
8x12	1386.987	0.003	0.121
8x13	891.353	15.004	0.074
8x14	4706.407	17.662	0.241
8x15	10444.553	7.551	1.511
9x1	0.102	0.076	0.078
9x2	0.487	2.542	2.621
9x3	7.489	0.136	0.002
9x4	14.718	0.443	0.003
9x5	17.237	0.018	0.003
9x6	26.867	0.380	0
9x7	73.422	0.010	0.001
9x8	172.684	3.500	0.019
9x10	1440.984	0.099	0.004
9x11	2184.552	7.480	0.240
9x12	4186.341	18.441	0
9x13	3879.812	22.099	0.002
9x14	18380.545	0.096	0.875
9x15	18413	73.336	0
10x1	0.079	0.071	0.073
10x2	0.518	2.408	2.204
10x3	10.011	0.018	0.001
10x4	29.268	0.058	0.006
10x5	14.485	0.421	0
10x6	133.029	0.539	0.060
10x7	102.510	0.053	0.019
10x8	1074.123	1.231	0
10x9	1250.916	6.209	0.014
10x11	3320.784	0.056	0.018
10x12	2894.197	0.901	0.316
10x13	7010.064	3.486	1.518
10x14	23393.980	0.051	0
10x15	10220.405	11.826	0.025
11x1	0.104	0.069	0.079
11x2	0.518	2.497	2.548
11x3	9.691	0.188	0.002
11x4	19.296	0.009	0.008
11x5	46.843	0.180	0.001
11x6	153.354	2.692	0.011
11x7	268.608	0.035	0.003
11x8	2122.992	3.554	0.309
11x9	1922.846	0.110	0.099
11x10	4199.389	21.520	0
11x12	14293.580	20.207	0.134
11x13	44841.441	17.432	5.617
11x14	76742.453	42.014	11.847
11x15	82497.438	912.064	7.784
12x1	0.118	0.073	0.078
12x2	0.593	2.841	2.908
12x3	12.845	0.015	0.001
12x4	3273.688	0.237	0
12x5	58.697	0.535	0.007
12x6	187.129	0	0.086
12x7	1621.575	2.010	0.008
12x8	1660.327	2.753	0.002
12x9	3551.272	25.219	0.006
12x10	10104.555	82.179	2.247
12x11	13300.496	24.414	0.039
12x13	96050.023	14.424	10.928
12x14	149051.344	882.036	0.015
12x15	659843.750	34.556	38.566
13x1	0.113	0.066	0.083
13x2	0.642	3.052	3.260
13x3	12.166	0.003	0
13x4	48.724	1.052	0
13x5	47.573	0.049	0.030
13x6	819.948	7.002	0.069
13x7	853.298	9.405	0.098
13x8	2528.936	0.648	0.485
13x9	5746.029	0.193	0.396
13x10	33875.543	195.350	3.348
13x11	58068.672	0	0.455
13x12	91959.719	53.041	0
13x14	181766.609	266.333	0
13x15	347118.250	83.535	43.105
14x1	0.100	0.088	0.114
14x2	0.739	3.173	3.297
14x3	17.111	0.003	0.007
14x4	77.127	1.537	0.002
14x5	64.599	2.595	0
14x6	1756.154	9.933	0.023
14x7	3417.269	0.120	0.307
14x8	8902.316	11.138	0.254
14x9	4927.492	1.967	0.393
14x10	46175.426	0.005	0.001
14x11	135851.422	603.407	0.001
14x12	124475.688	0.651	3.244
14x13	138244.531	1816.181	11.299
14x15	715761.312	1531.531	1.490
15x1	0.094	0.067	0.073
15x2	0.773	3.355	4.154
15x3	19.604	0.194	0.004
15x4	117.653	0.888	0.025
15x5	236.576	0.098	0.033
15x6	1106.127	5.051	0.031
15x7	1449.182	17.194	0.211
15x8	6642.612	0.064	0.709
15x9	20110.375	100.463	2.701
15x10	23095.273	1.598	6.475
15x11	94505.375	476.520	0.006
15x12	973994.125	11.600	11.664
15x13	862924.750	3825.050	11.389
15x14	1595494.750	7063.056	238.092

Table 6: Scores of player 1 playing cleverly with Random, Monte Carlo or Monte Carlo Tree Search against Dynamic Programming

From the above table, it becomes clear that for strategic methods, we need to differentiate between cases in which the first player was able to force a win and in which this player was not able to do so. When playing cleverly, the Monte Carlo Tree Search method is the method that performed best, as it obtained the highest scores of all methods most of the times. Although the Random method and Monte Carlo method occasionally obtained higher scores, this happened very few times and therefore, we ascribe this occurrence to chance. The second-best method in this case was the Monte Carlo method. Although the Random method produced better scores more often, these were only few results, in $3 \times n$-, $4 \times n$-, $5 \times n$- and $6 \times n$-boards. The results fluctuated a bit between the Random and Monte Carlo methods, as Random obtained better scores in $1 \times n$- and $m \times 1$-boards, but Monte Carlo obtained much better results for the other board-dimensionalities. Due to the large difference in scores, we believe that Monte Carlo performed better than Random. Interestingly, the Random method method was able to force wins more often in boards with three, four, five and six rows. This might indicate that there exist winning strategies for these boards that are yet unknown, as the known winning strategies that are incorporated in clever play did not force these wins for the other methods. The worst method then becomes Monte Carlo.

6.3 Experiment 3: Strategy stealing VS steal-any

For this experiment, we compare strategy stealing against the steal-any approach. We do not round up the scores to three decimals for boards that contain more than four rows as the scores become too small when the board-size increases. Below, the resulting table is provided.

In the table below, it is visible that in the vast majority of cases, strategy stealing tends to exhibit superior performance over the steal-any approach for smaller board-sizes. This means that in such board-sizes, in order to win the game, the first player should utilize strategy stealing, rather than the steal-any approach. Only rarely does the steal-any approach have the upper hand over strategy stealing. For medium-sized to larger board-sizes, the steal-any approach overall tended to obtain slightly higher scores, meaning that computationally, it performed slightly better than strategy stealing for these board-sizes.

Board	SS	SA
1x1	0	0
1x2	0.026	0.048
1x3	0.029	0.062
1x4	0.029	0.031
1x5	0.028	0.029
1x6	0.030	0.034
1x7	0.028	0.030
1x8	0.029	0.020
1x9	0.025	0.019
1x10	0.037	0.016
1x11	0.031	0.018
1x12	0.027	0.031
1x13	0.028	0.022
1x14	0.027	0.025
1x15	0.028	0.011
2x1	0.036	0.038
2x2	0.062	0.039
2x3	0.059	0.042
2x4	0.047	0.028
2x5	0.031	0.030
2x6	0.026	0.027
2x7	0.012	0.021—
2x8	0.020	0.017
2x9	0.017	0.018
2x10	0.012	0.016
2x11	0.013	0.014
2x12	0.014	0.018
2x13	0.012	0.012
2x14	0.011	0.010
2x15	0.010	0.011
3x1	0.041	0.027
3x2	0.048	0.049
3x3	0.032	0.026
3x4	0.027	0.026
3x5	0.017	0.019
3x6	0.016	0.010
3x7	0.009	0.010
3x8	0.007	0.008
3x9	0.005	0.006
3x10	0.006	0.005
3x11	0.005	0.005
3x12	0.003	0.003
3x13	0.003	0.003
3x14	0.002	0.002
3x15	0.002	0.002
4x1	0.020	0.018
4x2	0.031	0.028
4x3	0.031	0.016
4x4	0.021	0.014
4x5	0.008	0.009
4x6	0.011	0.006
4x7	0.005	0.004
4x8	0.004	0.004
4x9	0.002	0.003
4x10	0.001	0.002
4x11	0.001	0.001
4x12	0.001	0.001
4x13	0	0.001
4x14	0	0.001
4x15	0	0.001
5x1	0.0190683	0.0205498
5x2	0.0268355	0.0316715
5x3	0.0146638	0.0103828
5x4	0.0103389	0.0102833
5x5	0.0106354	0.00673267
5x6	0.00539047	0.0047993
5x7	0.00489154	0.0118556
5x8	0.00242761	0.00142556
5x9	0.000914697	0.000842669
5x10	0.00103539	0.000746397
5x11	0.000534927	0.000184142
5x12	0.00022847	0.00017859
5x13	0.000136073	0.00015612
5x14	0.000103491	5.58621e-05
5x15	0.000215177	4.71677e-05
6x1	0.0286632	0.023222
6x2	0.0176517	0.0332257
6x3	0.0132253	0.0106224
6x4	0.00996707	0.00660583
6x5	0.00432557	0.00388175
6x6	0.00243566	0.00270139
6x7	0.000349655	0.000373324
6x8	0.000623689	0.000460778
6x9	0.000533282	0.00034148
6x10	7.74387e-05	9.45905e-05—
6x11	0.000175697	8.97419e-05
6x12	4.00067e-05	6.9974e-05
6x13	5.25759e-05	6.10575e-05
6x14	2.73664e-05	2.9829e-05
6x15	3.71856e-05	3.99771e-05
7x1	0.0290491	0.0115731
7x2	0.0300415	0.0150908
7x3	0.0106309	0.00870345
7x4	0.00152247	0.00609038
7x5	0.00272986	0.00304559
7x6	0.00042262	0.00202907
7x7	0.000523543	0.000669848
7x8	0.000161842	0.000361993
7x9	0.000132258	0.000174608
7x10	0.000220978	8.80872e-05
7x11	3.97883e-05	6.12888e-05
7x12	2.78589e-05	2.79033e-05
7x13	2.45789e-05	2.59252e-05
7x14	1.35816e-05	1.40003e-05
7x15	7.44962e-06	0.00301543
8x1	0.035439	0.0143788
8x2	0.0152575	0.0173387
8x3	0.00741826	0.00876088
8x4	0.00322883	0.00277168
8x5	0.00289226	0.00166712
8x6	0.0002364	0.000267862
8x7	0.000174174	0.000345383
8x8	0.00016062	0.000259
8x9	0.000115509	0.000153831
8x10	4.79343e-05	5.78987e-05
8x11	1.81838e-05	2.4843e-05
8x12	1.04527e-05	1.15031e-05
8x13	1.26633e-05	1.38274e-05
8x14	6.76341e-06	8.36604e-06
8x15	1.96667e-06	2.229e-06
9x1	0.0348674	0.022749
9x2	0.0159768	0.0127806
9x3	0.000105165	0.00621297
9x4	0.00336221	0.00345493
9x5	0.000420788	0.0012915
9x6	0.000157805	0.000342409
9x7	0.000178632	0.000263442
9x8	0.000140938	0.000149432
9x9	1.96057e-05	2.54216e-05
9x10	3.60042e-05	4.18708e-05
9x11	1.07079e-05	1.14568e-05
9x12	1.05148e-05	2.21401e-05
9x13	1.07383e-06	2.64282e-06
9x14	1.59992e-06	2.11587e-06
9x15	1.43496e-06	1.5815e-06
10x1	0.0218985	0.0203271
10x2	0.0112191	0.0157929
10x3	4.79385e-05	0.00828354
10x4	0.00199288	0.00194094
10x5	0.00076801	0.000937743
10x6	0.000111149	0.000139658
10x7	5.70344e-05	0.000100675
10x8	5.41421e-05	5.80778e-05
10x9	3.43021e-05	4.47052e-05
10x10	8.77993e-06	1.03528e-05
10x11	3.96683e-06	4.86693e-06
10x12	3.12977e-06	8.38529e-06
10x13	1.33686e-06	2.71938e-06
10x14	1.12328e-06	1.36104e-06
10x15	3.69474e-07	4.61203e-07
11x1	0.0257016	0.0131949
11x2	0.0111587	0.00928319
11x3	0.00372036	0.00411417
11x4	0.00130836	0.00355089
11x5	0.000232425	0.000683545
11x6	0.000139187	0.000240443
11x7	5.62833e-05	6.26235e-05
11x8	3.02775e-05	2.95431e-05—
11x9	1.17156e-05	1.22293e-05
11x10	5.25209e-06	5.31835e-06
11x11	4.33304e-06	4.61787e-06
11x12	2.95769e-06	3.82545e-06
11x13	9.19698e-07	1.36576e-06
11x14	6.92316e-07	7.57042e-07
11x15	3.9587e-07	4.69301e-07
12x1	0.0274425	0.0201318
12x2	0.0103373	0.00872234
12x3	1.09066e-05	0.00250878
12x4	0.000837385	0.000863842
12x5	0.000543845	0.000589401
12x6	4.70836e-05	5.50765e-05
12x7	1.70849e-05	1.81611e-05
12x8	6.86432e-06	8.7046e-06
12x9	1.14636e-05	1.1905e-05—
12x10	3.10451e-06	3.81377e-06
12x11	2.68239e-06	2.71388e-06
12x12	4.75781e-07	6.47729e-07
12x13	7.94509e-07	9.55919e-07
12x14	1.22221e-07	1.30408e-07
12x15	1.31599e-07	6.58881e-07
13x1	0.0268804	0.0216058
13x2	0.0092414	0.00888613
13x3	0.00291624	0.00206444
13x4	0.000843228	0.000660549
13x5	0.000538752	0.000186793
13x6	4.00121e-05	7.49936e-05
13x7	2.53434e-05	2.70828e-05
13x8	1.48029e-05	1.14963e-05
13x9	2.84261e-06	2.97537e-06
13x10	7.97788e-07	6.58881e-06
13x11	8.11222e-07	8.22661e-07
13x12	5.56605e-07	7.28242e-07
13x13	8.01835e-08	1.64335e-07
13x14	5.59621e-08	6.02001e-08
13x15	1.74476e-08	9.11256e-08
14x1	0.0171198	0.0149545
14x2	0.0087939	0.0090209
14x3	0.00237967	0.00267141
14x4	0.000948379	0.000412466
14x5	5.22204e-05	6.41993e-05
14x6	4.09956e-05	3.62559e-05
14x7	1.55016e-05	1.6054e-05
14x8	1.25179e-05	1.30799e-05
14x9	2.4229e-06	4.70023e-06
14x10	1.17097e-06	1.81019e-06
14x11	3.80216e-07	1.07606e-06
14x12	1.61486e-07	1.64997e-07
14x13	3.08504e-08	6.0244e-08
14x14	3.54171e-08	7.07972e-08
14x15	6.15547e-08	9.22909e-08
15x1	0.0189688	0.00935741
15x2	0.00773905	0.00806869
15x3	0.00203201	0.00199295
15x4	0.000648363—	0.000537683
15x5	6.26327e-05—	8.11694e-05
15x6	0.000168296	3.89032e-05
15x7	8.13811e-06—	4.65308e-06
15x8	2.08652e-06	2.46548e-06
15x9	2.29238e-06	3.73851e-06
15x10	5.67457e-07	6.21745e-07
15x11	2.64071e-07	4.98094e-07
15x12	1.09067e-07	1.71161e-07
15x13	2.86266e-08	5.5141e-08
15x14	2.72084e-08	3.40248e-08
15x15	6.44985e-09	1.00353e-08

Table 7: Scores of Strategy Stealing (SS) and Steal-any (SA) against Dynamic Programming

7 Conclusions and Further Research

In conclusion, the game of Chomp is a game that lends itself to broad mathematical and computational analysis. We made a step forward by enhancing the knowledge about this game, as well as by clarifying the relation between this game and the game of Nim. It turns out that this game is directly translatable to Nim and vice versa. Moreover, we enhanced the knowledge about the nature of Chomp by means of computational analysis over various methods for the first player. We found that when comparing the methods between each other, Monte Carlo performed the best, Monte Carlo Tree Search was the next best and Random performed worst. It turns out that, in case the adversary plays perfectly, the first player can best select moves using Monte Carlo, when he has no knowledge of winning properties of board-states. The Monte Carlo method is also better at finding winning moves for this player. In case the first player has knowledge of winning properties, then he can choose to play cleverly. In that case, the Monte Carlo Tree Search method proved itself to be the most worthwhile of all methods. Finally, it turned out that strategy stealing exhibited superior performance over the steal-any approach for smaller boards and the steal-any approach exhibited superior performance for medium-sized and larger boards. Thus, for the first player to win, it is best to either apply a winning strategy if one is known, for example, in the case of game states that constitute winning properties, or to apply strategy stealing in the case of small board dimensions and the steal-any approach in the case of medium to large board-dimensions.

7.1 Future work

For future research, it would be worthwhile to analyze what the effect of increasing the number of players would be on first-player wins and on the probability of the first player being able to win. Although a game with more than two players is no longer combinatorial, the analysis of it is still valuable, because the strategies that can be deployed are different: for a player to win, this entails forcing an arbitrary possible other player to lose instead of strictly the second. Hence, many strategies that are applicable in two-player Chomp will no longer be applicable in multiplayer Chomp, and it would be interesting to find out what (kind of) strategies are consonant with multiplayer Chomp. Moreover, in two-player Chomp, it would be interesting to do research on which Chomp-areas and -shapes tend to be more winning in general, for different board-dimensions, by means of calculations of winning probabilities per move. This can be done by selecting a move and performing a specified number of simulations from thereon, assigning a score to the performed move, and undoing the move. Eventually, all legal moves would then have scores assigned and various area's can get general average scores. Hereby, game-boards can be into heatmaps, where the hotspots are the areas that have the highest probability of winning. This idea could in fact aid with the analysis of player wins in multiplayer Chomp, hence, for future research, a combination of both could be profound. Most notably, the game of Slicing Chomp is a novel combinatorial game that lends itself to thorough investigation by means of the Sprague-Grundy theorem. As such, investigating it may prove worthwhile.

References

[1] E. R. Berlekamp, J. H. Conway, and R. K. Guy. Second edition. *Winning Ways for Your Mathematical Plays*, 1:40, 1982.

[2] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3:35–39, 1901.

[3] C. Browne, E. Powley, D. Whitehouse, S. 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*, 4:1–2, 4–8, 2012.

[4] T. S. Ferguson. Impartial combinatorial games. *Game Theory*, I:3–5, 9–11, 14–15, 21–22, 40–44, 2000.

[5] T. S. Ferguson. Two-person zero-sum games. *Game Theory*, II:45–46, 2000.

[6] D. Gale. A curious Nim-type game. *The American Mathematical Monthly*, 81:876–879, 1974.

[7] M. Gardner. Sim, Chomp and Race Track. *Knotted Doughnuts and Other Mathematical Entertainments*, Chapter 9:119–120, 1986.

[8] T. Lattimore and C. Szepasvári. The upper confidence bound algorithm: Bernoulli noise. *Bandit Algorithms*, pages 133–134, 137, 2020.

[9] E. H. Moore. A generalization of the game called Nim. *Annals of Mathematics*, 11:93, 1910.

[10] B. Muller. Quarto. *https://en.gigamic.com/modern-classics/1049-quarto-3421271300410.html*, 1991.

[11] T. Patil. Combinatorial games. *Mathematik-Olympiade*, Version 1.0.1:3–4, 10–11, 2023.

[12] A. Rabah and I. V. Evstigneev. On Zermelo's theorem. pages 1–4, 2016.

[13] F. Schuh. Spel van de delers. *Nieuw tijdschrift voor de wiskunde*, 39:299–304, 1951.

[14] M. H. M. Winands. Monte-Carlo Tree Search. *Encyclopedia of Computer Graphics and Games*, Arxiv:1–3, 2015.

[15] W. H. Wythoff. A modification of the game of Nim. *Nieuw Archief voor Wiskunde*, 7:199, 1907.