A Program for Analysing Combinatorial and Synchronized Games

Lenstra, A.H.K.

Universiteit Leiden
The Netherlands

Mathematics & Computer Science

A Program for Analysing Combinatorial and Synchronized Games

Xander Lenstra

Supervisors:
Walter Kosters
Mark van den Bergh

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Mathematical Institute, Leiden University (MI)
www.liacs.leidenuniv.nl

31 July 2022

Abstract

In this thesis, we will the basic concepts of Combinatorial Game Theory and Synchronized Game Theory. We introduce CGSynch, a program we have written to analyze games from these fields, and explain how this program works. Lastly, we show the correctness of CGSynch and compare CGSynch with CGSuite, the currently best application for analyzing combinatorial games.

Contents

1 Introduction

Combinatorial Game Theory is a subfield of Game Theory, in which a subset of games is analyzed that are in some sense simple. That is, two-player games that are deterministic, in which both players have perfect information and take turns making moves. Despite this, the theory of these games is surprisingly rich and analyzing some of these games, especially larger ones, is non-trivial. An application, CGSuite, was written to aid in making these analyses.

Synchronized Game Theory is a related field, in which a similar but different class of games is analyzed. In these deterministic two-player games both players make their moves simultaneously, and thus players in some sense no longer have perfect information. As a result, synchronized games are even more difficult to analyze by hand. However, CGSuite cannot analyze these games, and also no similar application existed for this class of games. Therefore, most analyses of these games had to be done by hand, using mathematical tricks or using ad-hoc written computer programs.

To solve this, we have written a new application, CGSynch. This program was written to determine simple properties of both combinatorial games and synchronized games. While in principle the user interface is focused on inputting positions of Push, Shove, Cherries, Stack Cherries and Hackenbush, the code was made with extensibility in mind and as such adding other games is certainly possible. The code for this program can be found on GitHub [10], including documentation on how to extend its analysis to other games.

In Section 2 of this thesis, we will explain the basics of Combinatorial Game Theory, and in Section 3 we will explain the basics of Synchronized Game Theory. Both of these sections will contain definitions and theorems that were used in CGSynch, as well as explanation of how the combinatorial and synchronized versions of the games Hackenbush, Push, Shove, Cherries and Stack Cherries are played. In Section 4 we will give a high-level explanation of how CGSynch works. In Section 5 we will show that CGSynch can accurately determine properties of combinatorial and synchronized games. We will also show that for combinatorial games CGSynch is faster than CGSuite, at least for computing the canonical form of games.

This thesis has been written at Leiden University as part of the double bachelor of mathematics and computer science. It was supervised by Walter Kosters and Mark van den Bergh.

2 Combinatorial Games

This section will contain definitions and theorems for and about combinatorial games. Most of the proofs of these theorems are omitted for brevity, but all of them can be found in [1, 4, 12].

Combinatorial games are deterministic games where two players alternate taking turns and have perfect information of the state of the game. The first player unable to make a move loses. We start by looking at examples of such games.

2.1 Rulesets

There are many different combinatorial games one can play, but the application that we have created for analyzing combinatorial games supports only five of them. The rulesets describing these games, Hackenbush, Cherries, Stack Cherries, Shove and Push, are therefore explained in this section. We will also use positions of these rulesets as examples in the rest of this thesis.

For other examples of combinatorial games, we refer the reader to [1, Appendix 4], which lists a large number of rulesets.

A position in Hackenbush is a normal graph in which a single node is labelled as "the ground", and in which all edges are coloured blue or red. The labelled node is often drawn as a horizontal line with its adjacent edges rooted in different places. An example of such a position can be seen in Figure 2.1.

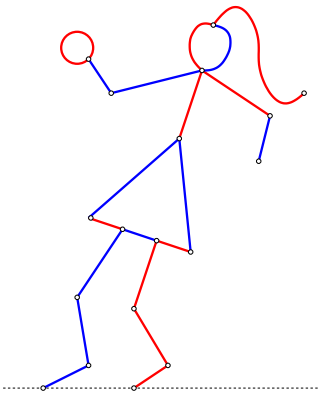


Figure 2.1: An example of a Hackenbush position [13].

Two players, Blue and Red, take turns removing an edge of their colour. If after their move any part of the graph becomes disconnected from the ground, that part of the graph is also removed from play.

A variant of this game allows for some edges to be green, indicating that they may be removed by either player. The application we have written does not support these green edges, but we mention this here as it is used for an example later on.

The game Cherries is played on a strip of finite length. Each square in this strip can either contain a black stone, a white stone or no stone at all. An example of such a position can be seen in Figure 2.2.

Figure 2.2: An example of a CHERRIES position.

On their turn, each player may remove any stone if it is of their colour and it is adjacent to either an empty square or either end of the strip.

The game STACK CHERRIES is very similar to CHERRIES in that it is also played on a finite strip in which each square contains either a black or white stone or no stone at all. Additionally, players may still only remove a stone of their own colour. However, this stone must now specifically have an empty square to the left of it, or be the left-most stone of the strip. In some sense STACK CHERRIES is thus a restricted version of CHERRIES.
An example of such a position can be seen in Figure 2.3. The triangle on the left is used to distinguish STACK CHERRIES positions from CHERRIES positions.

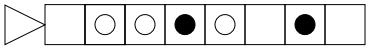


Figure 2.3: An example of a STACK CHERRIES position.

The game PUSH is also played on a finite strip. Instead of black and white stones, the squares in this strip can contain either a blue or a red hand. An example of such a position can be seen in Figure 2.4.

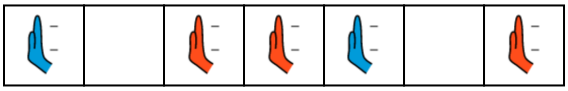


Figure 2.4: An example of a PUSH position.

On their turn, each player may choose a hand of their own colour. This hand and every hand left of it up to the next empty square or the edge of the board is moved one square to the left. Any hand that is pushed or moved off of the strip is removed from play.

The game SHOVE is another game that is played on a finite strip. Just as in PUSH, the squares of this strip contain a red or blue hand, but with more "windy" lines drawn behind it. An example of such a position can be seen in Figure 2.5.

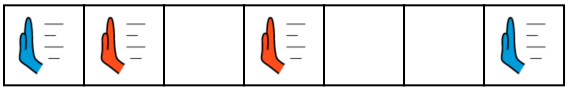


Figure 2.5: An example of a SHOVE position.

On their turn, each player may choose a hand of their own colour. That hand and every piece to the left of it are moved one square to the left, even if there are one or more empty squares in between it and the chosen piece. If this would move a piece off of the strip, it is instead removed from play.

2.2 Games and Outcome Classes

Combinatorial games are generally the same as those discussed above. There are two players, Left, Blue or Black, who is usually female, and Right, Red or White, who is usually male. These genders allow us to refer to players as "she" or "he" and make it immediately clear who is referred to. These players take turns making moves. When after a finite number of turns one player is unable to move, this player is declared the loser.

Definition 2.1. A *combinatorial game* G is a pair $\{\mathcal{G}^L \mid \mathcal{G}^R\}$, where $\mathcal{G}^L$ and $\mathcal{G}^R$ are sets of combinatorial games.

These sets $\mathcal{G}^L$ and $\mathcal{G}^R$ are also called the option sets of Left and Right, respectively, and contain the positions either player can move to from this position.

Example 2.2. An example of the definition of a HACKENBUSH game written as a combinatorial game can be seen in Figure 2.6.

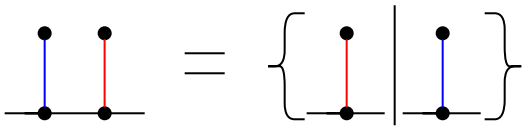


Figure 2.6: An example of the definition of a Combinatorial Game.

◁

The class of all combinatorial games is very large, and in this thesis we are only interested in a certain subclass of these. We will define this subclass $\mathbb{G}$ in the following definition.

Definition 2.3. We set $\mathbb{G}_0 = \{\{\emptyset \mid \emptyset\}\}$, the set only containing the empty game.

For all $n \in \mathbb{N}_0$, we recursively define $\mathbb{G}_{n+1} = \{\mathcal{G}_n^L \mid \mathcal{G}_n^R\}$ where $\mathcal{G}_n^L, \mathcal{G}_n^R \subseteq \mathbb{G}_n$.

Lastly, we set $\mathbb{G} = \bigcup_{n \in \mathbb{N}_0} \mathbb{G}_n$.

By only looking at combinatorial games contained in $\mathbb{G}$, we restrict ourselves to looking at games that end after a finite number of moves. Forgoing this assumption can lead for example to *loopy* games, in which a position can be reached from itself after one or more moves, or *transfinite* games, which have can take an infinite amount of moves to play out. These classes of games have their own very interesting mathematical properties, but we will not focus on these in this thesis. The reader that is interested in these, can read more about them in [1, Appendix 4] or in [12].

Next, we define the birthday of a combinatorial game.

Definition 2.4. Let $G = \{\mathcal{G}^L \mid \mathcal{G}^R\} \in \mathbb{G}$ be a combinatorial game. We recursively define its *birthday* $b(G)$ by:

$$b(G) = \begin{cases} 0 & \text{if } \mathcal{G}^R = \mathcal{G}^L = \emptyset, \\ 1 + \max\limits_{H \in \mathcal{G}^L \cup \mathcal{G}^R} b(H) & \text{else.} \end{cases}$$

Equivalently, one can say that the birthday of G is the smallest $n \in \mathbb{N}$ such that $G \in \mathbb{G}_n$, or that the birthday of G is the height of its the game tree.

We can subdivide this set $\mathbb{G}$ based on the *outcome class* of a combinatorial game G, that is, the winner of the game G assuming perfect play from both players. This results in four classes $\mathcal{L}, \mathcal{R}, \mathcal{N}, \mathcal{P} \subset \mathbb{G}$ as shown in Table 2.1.

		Right moves first	
		Left wins	Right wins
Left moves first	Left wins	$\mathcal{L}$ (left)	$\mathcal{N}$ (next)
	Right wins	$\mathcal{P}$ (previous)	$\mathcal{R}$ (right)

Table 2.1: The winner of a position based on who wins and who starts playing.

Example 2.5. In Figure 2.7, examples of positions belonging all winning classes are shown.

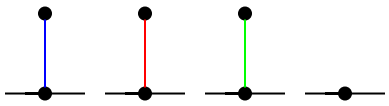


Figure 2.7: Examples of Hackenbush positions from winning classes $\mathcal{L}$, $\mathcal{R}$, $\mathcal{N}$ and $\mathcal{P}$ respectively.

◁

We define $o\colon \mathbb{G} \to \{\mathcal{L}, \mathcal{R}, \mathcal{N}, \mathcal{P}\}$ to be the function that maps a game position to its outcome class. This value can be found recursively using the following theorem.

Theorem 2.6. Let $G = \{\mathcal{G}^L \mid \mathcal{G}^R\} \in \mathbb{G}$ be some game. Then the outcome class $o(G)$ can be determined recursively by using Table 2.2.

	$\exists G^R \in \mathcal{G}^R$ such that $G^R \in \mathcal{R} \cup \mathcal{P}$	$\forall G^R \in \mathcal{G}^R$: $G^R \in \mathcal{N} \cup \mathcal{L}$
$\exists G^L \in \mathcal{G}^L$ such that $G^R \in \mathcal{L} \cup \mathcal{P}$	$o(G) = \mathcal{N}$	$o(G) = \mathcal{L}$
$\forall G^L \in \mathcal{G}^L$: $G^L \in \mathcal{N} \cup \mathcal{R}$	$o(G) = \mathcal{R}$	$o(G) = \mathcal{P}$

Table 2.2: Table defining the outcome classes of games.

Next, we will define a partial order on the set of outcome classes by $\mathcal{L} > \mathcal{P}, \mathcal{N}$ and $\mathcal{P}, \mathcal{N} > \mathcal{R}$. This will be used in Section 2.4 when defining the partial order of combinatorial games.

2.3 Addition and Subtraction

In this section we will define an addition operation $+$ on $\mathbb{G}$. Later in this section, we will define its inverse operation $-$.

The addition operation is defined below. Intuitively, we see $G + H$ as a combinatorial game in which each player is allowed to move on either G or H, leaving the other operand as is.

Definition 2.7. Let $G, H \in \mathbb{G}$, with $G = \{\mathcal{G}^L \mid \mathcal{G}^R\}, H = \{\mathcal{H}^L \mid \mathcal{H}^R\}$. Then we recursively define:

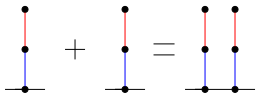
$$G + H = \{\mathcal{G}^L + H, G + \mathcal{H}^L \mid \mathcal{G}^R + H, G + \mathcal{H}^R\}.$$

Here, addition with a set is defined as the union of the sum of each element, for example:

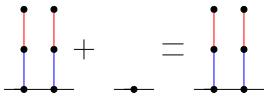
$$\mathcal{G}^L + H = \bigcup_{G^L \in \mathcal{G}^L} \left\{G^L + H\right\}.$$

We also define the game $0 = \{\emptyset \mid \emptyset\}$, which is a neutral element of this addition.

Example 2.8. Two examples of addition of two HACKENBUSH positions can be found in Figure 2.8.



(a) A sum of two HACKENBUSH positions.



(b) A sum of a HACKENBUSH position and 0.

Figure 2.8: Examples of sums of HACKENBUSH positions.

◁

Using this definition of addition, we can also define the inverse of a game. Intuitively, this will be the game in which Right has all the options that Left had in the original game and vice versa.

Definition 2.9. Let $G \in \mathbb{G}$ with $G = \{\mathcal{G}^L \mid \mathcal{G}^R\}$. Then we recursively define:

$$-G = \{-\mathcal{G}^R \mid -\mathcal{G}^L\}.$$

Where the inverse of a set is defined as the set of the set of the inverses, for example:

$$-\mathcal{G}^L = \bigcup_{G^L \in \mathcal{G}^L} \left\{-G^L\right\}.$$

Next, we define equality on games:

Definition 2.10. Let $G, H \in \mathbb{G}$. Then:

$$G = H \iff \bigvee_{X \in \mathbb{G}} o(G + X) = o(H + X).$$

Theorem 2.11. The $=$ defined above is an equivalence relation.

It also turns out there is an easy way to check whether a game is equal to 0:

Theorem 2.12. Let $G \in \mathbb{G}$. Then $G = 0 \iff G \in \mathcal{P}$.

While the equality defined above is very useful, it is sometimes also useful to denote when two games are exactly the same. That is, their left option sets are equal and their right option sets are equal. We define this below.

Definition 2.13. Let $G, H \in \mathbb{G}$ be two combinatorial games. We say G and H are isomorphic and write $G \cong H$ if and only if $\mathcal{G}^L \cong \mathcal{H}^L$ and $\mathcal{G}^R \cong \mathcal{H}^R$.

It turns out the addition and subtraction we have defined have all the natural properties one would expect from an addition and subtraction and they give rise to an Abelian group $(\mathbb{G}, 0, +, =)$. One can find more information on this in [1].

2.4 Comparing Games

In the previous section we defined an addition and subtraction on the set $\mathbb{G}$ and noted that these have natural properties resulting in $\mathbb{G}$ being an Abelian group. In this section we will define a partial order on $\mathbb{G}$.

Definition 2.14. Let $G, H \in \mathbb{G}$. Then $G \geq H$ if for all $X \in \mathbb{G}$: $o(G + X) \geq o(H + X)$.

Intuitively, this definition says that whenever we come across a position of which H is a summand, replacing this with G only improves the original position for Left.

We define $>$, $<$ and $\leq$ as one would expect based on the definition of $\geq$.

Theorem 2.15. The relation $\geq$ is a partial ordering on $\mathbb{G}$.

Furthermore, the outcome class of a game is closely linked to how it compares to the game 0.

Theorem 2.16. Let $G \in \mathbb{G}$. Then:

$$\begin{aligned} G \geq 0 &\iff G \in \mathcal{L} \cup \mathcal{P} \\ G > 0 &\iff G \in \mathcal{L} \\ G < 0 &\iff G \in \mathcal{R} \end{aligned}$$

Based on the above theorem and intuition on how the ordering of real numbers work, one might expect that all games are either equal to, or smaller or greater than 0, and might conclude that there are no games in which the next player wins. This is, however, not the case, because there exist combinatorial games that cannot be compared with 0. An example of this is the Hackenbush position $\{0 \mid 0\}$ in Example 2.7. For such games, the following notation is used.

Definition 2.17. Let $G, H \in \mathbb{G}$, such that neither $G \leq H$ nor $H \leq G$. We then say G and H are incomparable, and write $G \parallel H$.
Additionally, we write $G \mathrel{|\!\rhd} H$ when either $G \parallel H$ or $G > H$, and similarly for $G \lhd\!| H$.

Using this, we can find the games where the next player wins.

Theorem 2.18. Let $G \in \mathbb{G}$. Then:

$$G \parallel 0 \iff G \in \mathcal{N}$$

2.5 Canonical Form

It is often possible to simplify combinatorial games. For example, if $G = \{\mathcal{G}^L \mid \mathcal{G}^R\}$ is a combinatorial game such that for some $G^L, G^{L'} \in \mathcal{G}^L$ we have that $G^{L'} < G^L$, the left player will never play to the position $G^{L'}$ from G. In some sense, they can thus be removed from $\mathcal{G}^L$ without changing the game that is being played. We then say that $G^{L'}$ is a *dominated* position, or that $G^{L'}$ is *dominated* by G^L.

Theorem 2.19. Let $G = \{A, B, C, \ldots \mid \alpha, \beta, \gamma, \ldots\} \in \mathbb{G}$ be such that $B \geq A$ and $\beta \leq \alpha$. Then $G = \{B, C, \ldots \mid \beta, \gamma, \ldots\}$.

Apart from domination, there is another way to simplify a combinatorial game: *reversibility*.

Theorem 2.20. Let $G = \{A, B, C \ldots \mid \alpha, \beta, \gamma, \ldots\} \in \mathbb{G}$ be such that there exists a $A^R \in \mathcal{A}^R$ such that $G \geq A^R$. Write $A^R = \{A', B', C', \ldots \mid \alpha', \beta', \gamma', \ldots\}$. Then by *reversibility* we have that $G = \{A', B', C', \ldots, B, C, \ldots \mid \alpha, \beta, \gamma, \ldots\}$.

Example 2.22 shows an example of reversibility. Of course, reversibility also applies to the other player, that is, if all options of both players are swapped and there exists an $\alpha^L \geq G$ with $\alpha^L = \{A', B', C', \ldots \mid \alpha', \beta', \gamma'\}$ then $G = \{A, B, C, \ldots \mid \beta, \gamma, \ldots, \alpha', \beta', \gamma' \ldots\}$.

We thus have found two ways of simplifying a position. By starting with any game G, we can always apply these simplifications a finite number of times until no more simplifications are possible. We then say that this new game G' is in *canonical form* and that it is the *canonical form* of G.

One might ask whether the canonical form of a position is unique, and this turns out to be the case.

Theorem 2.21. Let $G, G' \in \mathbb{G}$ be in canonical form such that $G = G'$. Then $G \cong G'$.

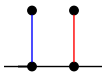


Figure 2.9: A HACKENBUSH position won by the previous player.

Example 2.22. Let G be the combinatorial game corresponding to the HACKENBUSH position in Figure 2.9, that is, $G = \{-1 \mid 1\}$. Then the canonical form of G is the game 0.

One can compute this by noticing that both the -1 and 1 are reversible, as they have a left respective right option of $\{ \mid \} = 0$ and $0 \geq \{-1 \mid 1\}$ as $0 - \{-1 \mid 1\}$ is a win for Left playing second. Replacing these options with the left respective right options of 0 results in the game $\{ \mid \} = 0$, which cannot be simplified further. Thus, $G = 0$ as expected based on Theorem 2.12. ◁

2.6 Values of Games

In the previous sections, we defined what a combinatorial game was, and showed that $\mathbb{G}$ is a partially ordered Abelian group. We also defined a game 0, which already seems to imply that we can assign numbers to other games. This indeed turns out to be the case.

Definition 2.23. Let $n \in \mathbb{N}_{>0}$. Then we define the following games:

$$n = \{n-1 \mid \}$$
$$-n = \{ \mid -n+1\}$$

Furthermore, these numbers behave under the operations defined before as one would expect.

Theorem 2.24. Let $A, B, C \in \mathbb{G}$ with respective values $a, b, c \in \mathbb{Z}$. Then:

$$A \geq B \iff a \geq b$$
$$A = -B \iff a = -b$$
$$A + B = C \iff a + b = c$$

As a result of these statements, integers are isomorphic with a subset of $\mathbb{G}$. We will use this isomorphism to interchangeably use integers and games with integer value when referring to them. For example, '13' will be used for both the game $\{12 \mid\}$, whose value is 13, and for the integer 13 itself.

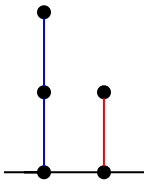


Figure 2.10: A HACKENBUSH position with value 1.

Example 2.25. In Figure 2.10 one can see a HACKENBUSH position with value $2 + -1 = 1$. ◁

We can also define a subset of fractions, usually called *the dyadic fractions*. These are all the fractions with a power of two as their denominator.

Definition 2.26. Let $m, j \in \mathbb{Z}$ with m an odd number and $j \geq 1$. Then we define the following game:

$$\frac{m}{2^j} = \left\{ \frac{m-1}{2^j} \,\middle|\, \frac{m+1}{2^j} \right\}$$

Just like integers, these games behave as one would expect under the operations we defined before. That is, Theorem 2.24 still holds when we take the values of games A and B to be either integers or dyadic fractions. In general, we will call a combinatorial game that is either an integer or a dyadic fraction a *number*.

By using Theorem 2.24 we can see that numbers are in many cases easier to work with than general combinatorial games. Additionally, we can in some cases easily determine the value of a game whose options are all numbers.

Definition 2.27. If x^L, x^R are numbers and $x^L < x^R$, then we define the simplest number x between x^L and x^R as follows:

- If there exists an integer $n \in \mathbb{Z}$ such that $x^L < n < x^R$, then we take x to be the smallest such number in absolute value.
- Otherwise, we take x to be the dyadic fraction $\frac{m}{2^j}$ such that $x^L < x < x^R$ for which j is minimal.

Theorem 2.28 (Simplest Number Theorem)**.** Let $G \in \mathbb{G}$ be such that all left and right options are numbers, and such that all left options are smaller than all right options. Then G is equal to the simplest number between the maximal left option and the minimal right option.

Theorem 2.29. Let $G = \{\mathcal{G}^L \mid \mathcal{G}^R\} \in \mathbb{G}$. If there exists a number x such that $\mathcal{G}^L \lhd\rhd x \lhd\rhd \mathcal{G}^R$, then $G = x$.

With this last theorem, we conclude our exploration of the basics of Combinatorial Game Theory.

2.7 Efficiently Computing Values of Shove and Cherries games

In the previous section, we have seen how we can calculate the value of many different combinatorial games. However, when we analyze combinatorial games from a specific ruleset, we can often use properties of that ruleset to simplify this calculation. In this section, we will note some theorems on the values of Shove and Cherries positions.

We will first give a simple algorithm to compute the value of any Cherries position from [11]. A Cherries position consists of a strip of white stones, black stones and empty squares. Furthermore, if there is ever an empty square in the strip, no moves to the left of it will have any effect on the position to the right of it and vice versa. We can thus split the strip at each empty square. Summing the segments obtained in this way results in a game equal to the original. For example, see the equality in Figure 2.11.

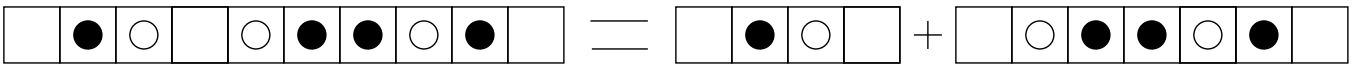


Figure 2.11: An example of splitting a Cherries position into segments.

As a result, it is sufficient to determine the value of a position only containing empty squares on the edges of its strip, as summing the values of these segments will result in the value of the original entire strip.

Next, we further subdivide these segments into blocks, each maximal in size and consisting of stones of only a single colour. For example, the right segment of the position in Figure 2.11 has four blocks: a block of one white stone, a block of two black stones, etc. We say that the *sign* of a block is 1 if the block consists of black stones, and it is -1 if the block consists of white stones. Furthermore, we call a block *internal* if it does not touch any empty squares.

Theorem 2.30. Let S be a Cherries position consisting only of a single segment. Let m be the length of the first block of Cherries of S, and n the length of the last block. Let ℓ, r be the signs of these blocks respectively. If the segment contains any internal block with a length strictly

greater than 1, let then x, y be the signs of the leftmost and rightmost internal blocks, which may coincide. Otherwise, set $x = y = 0$. Then the value of S is equal to:

$$\ell(m-1) + r(n-1) + \frac{\ell + r}{2} + \frac{x + y}{2}.$$

Example 2.31. The value of the left segment in Figure 2.11 is

$$-1(0) + 1(0) + \frac{1-1}{2} + \frac{0+0}{2} = 0.$$

The value of the right segment in Figure 2.11 is

$$-1(0) + 1(1) + \frac{1-1}{2} + \frac{1+1}{2} = 2.$$

◁

We can also determine the value of any SHOVE position by similarly analyzing the position instead of computing the game tree. The method below is proven in [1].

We only look at a single strip, as for any position consisting of multiple strips we can determine the value by summing the values of each of the strips. Next, let n be the number of hands in this strip. We define two functions $p\colon \{1, \ldots, n\} \to \mathbb{N}$ by $p(i)$ the position of the i'th hand from the left, and $c\colon \{1, \ldots, n\} \to \{\pm 1\}$ by $c(i)$ the sign of the colour of the i'th hand form the left, were red hands have value -1 and blue hands have value 1. We also define the function $r\colon \{1, \ldots, n\} \to \mathbb{N}$ by $r(i)$ the number of hands to the right of hand i up to and including the last alternation of colour. An example of this can be seen in Figure 2.12.

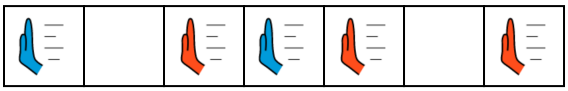


Figure 2.12: The values of r for the hands above are 3, 2, 1, 0 and 0 respectively.

Theorem 2.32. Let S be some SHOVE strip with n hands on it. Then the value of S is equal to:

$$\sum_{i=1}^{n} c(i) \frac{p(i)}{2^{r(i)}}.$$

Example 2.33. The value of the strip in Figure 2.12 is

$$\frac{1}{8} - \frac{3}{4} + \frac{4}{2} - 5 - 7 = -10\frac{1}{4}.$$

◁

3 Synchronized Games

In the previous section, we have discussed combinatorial games, their properties, and a few rulesets of games that can be modelled as combinatorial games. In this section, we will discuss a related class of games, synchronized games.

The most important difference between combinatorial games and synchronized games is that the moves of both players are performed simultaneously instead of successively, as is done in combinatorial games. Unlike in combinatorial games, a game can now also end when both players have no moves left. In that case, we say the game ended in a draw.

By making small changes to the rulesets defined in Section 2.1, we can turn them into synchronized games. In this section, we will first discuss these changes, give a general definition of synchronized games and then list a few properties of synchronized games. Most of these definitions and theorems can be found in more detail in [3].

3.1 Changes to Rulesets

We will use the same visuals when discussing positions of synchronized rulesets as we do for the combinatorial versions of these rulesets. From the context it should be clear whether we are talking about the combinatorial or synchronized versions of a ruleset.

Synchronized Hackenbush is played the same as combinatorial Hackenbush, except both players simultaneously choose an edge. These edges are removed, and only then all nodes and edges no longer connected to the ground are removed from play.

The only difference between combinatorial Cherries and Synchronized Cherries is that the stones chosen by both players are removed at the same time. The same is true for Synchronized Stack Cherries.

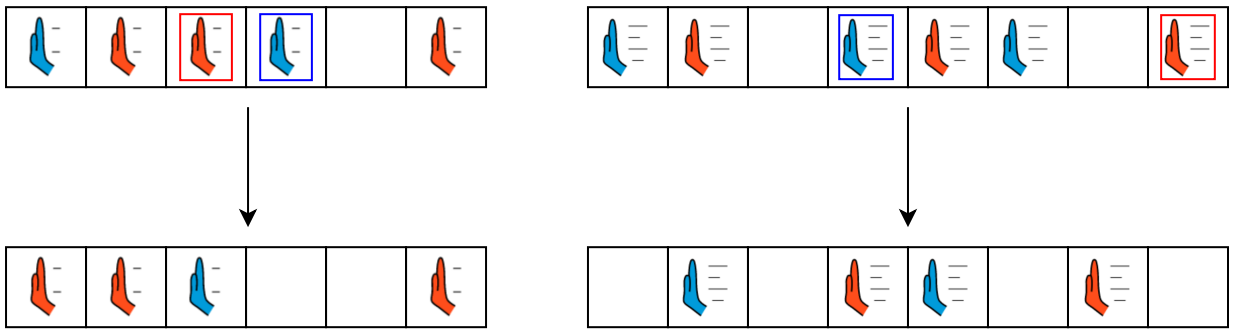
In Synchronized Shove, both players choose a hand of their own colour. The left-most of these is moved first and then the right-most, both as they would normally be moved in Shove. As a result, the left-most selected hand and all hands in front of it are moved two squares or off of the strip, and all hands between the two selected hands are moved one square.

Similarly, in Synchronized Push, after both players have chosen a hand, the left-most is moved first as it would be in combinatorial Push. Then the right-most hand is moved. While in Synchronized Shove it is possible for a hand to move two squares, this is thus not possible in Synchronzied Push.

Example 3.1. An example of a move in Synchronized Push and Shove can be seen in Figure 3.1. ◁

3.2 Basics of Synchronized Game Theory

It is clear that, in many ways, synchronized games and combinatorial games are very similar. However, because now both Left and Right make a move simultaneously, we can no longer model it using just two sets of options for the Left and Right player. This can be seen in the following definition.



(a) An example of a move in SYNCHRONIZED PUSH. (b) Example of a move in SYNCHRONIZED SHOVE.

Figure 3.1: Examples of moves in two synchronized games; the outlined pieces are the pieces each player moves.

Definition 3.2. A *synchronized game* is a triple $(\mathcal{G}^L, \mathcal{G}^R, \mathcal{G}^S)$. Here, $\mathcal{G}^L$ and $\mathcal{G}^R$ are sets of synchronized games, and $\mathcal{G}^S$ is a $|\mathcal{G}^L| \times |\mathcal{G}^R|$ matrix, in which each entry is a synchronized game.

In general, we interpret these sets as follows: $\mathcal{G}^L$ is the set, or, more formally, sequence, of left options, that is, games reached if *only* Left would make a move. Similarly, $\mathcal{G}^R$ is the set or sequence of right options. Then we create the matrix $\mathcal{G}^S$ by taking each element $\mathcal{G}^S_{ij}$ to be the position in which the i'th move of $\mathcal{G}^R$ is performed and the j'th move of $\mathcal{G}^L$ is performed.

Similarly to what we have done for combinatorial games, we will define the birthday of a synchronized game.

Definition 3.3. Let $G = (\mathcal{G}^L, \mathcal{G}^R, \mathcal{G}^S)$ be a synchronized game. Then we define the birthday of G to be the result of the function b given by:

$$b(G) = \begin{cases} 0 & \text{if } \mathcal{G}^S \text{ is empty,} \\ 1 + \max_{i,j} b(\mathcal{G}^S_{ij}) & \text{else.} \end{cases}$$

If b would be ill-defined for this game G, for example when G is contained in the matrix $\mathcal{G}^S$, we instead set $b(G) = \infty$.

We restrict ourselves to looking solely at games with finite birthday. We use $\mathbb{S}$ to denote the set of all synchronized games with a finite birthday. This is in a sense similar to our restriction of combinatorial games to the set $\mathbb{G}$.

One might ask why we need the sets $\mathcal{G}^L$ and $\mathcal{G}^R$, if all positions that can be reached by both players making a move are contained in $\mathcal{G}^S$. One reason for this is for determining the winner of a game.

Definition 3.4. We call a synchronized game $G = (\mathcal{G}^L, \mathcal{G}^R, \mathcal{G}^S)$ *decided* if G^S is empty.

There are three possible reasons why this could be the case, and each of these also defines which player has won the game. If $\mathcal{G}^L = \emptyset$ but $\mathcal{G}^R \neq \emptyset$, we declare Left the winner. If $\mathcal{G}^R = \emptyset$ but $\mathcal{G}^L \neq \emptyset$, we declare Right the winner. However, if $\mathcal{G}^L = \mathcal{G}^R = \emptyset$, that is, both players have no moves left, then we declare the game to have ended in a draw. Similar to how it is done for combinatorial games, we will denote these outcome classes with $\mathcal{L}, \mathcal{R}$ and $\mathcal{D}$, respectively.

It turns out that, unlike combinatorial games, synchronized games are not necessarily part of only one of these outcome classes.

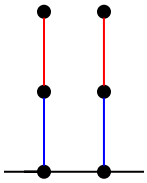


Figure 3.2: A simple HACKENBUSH position.

Example 3.5. Let $G \in \mathbb{S}$ correspond to the HACKENBUSH position in Figure 3.2. Both players have to make a choice to play on either the left or the right stalk. If they choose to play on the same stalk, only a single stalk will remain with an edge of each colour. Both players must play on the edge of their colour, after which no edges are left and the game ends in a draw. However, if they play on different stalks on their first move, only a single blue edge will remain, and the game ends in a win for Left. As both players do not know the move the other player makes, there is no way for Right to know what move he should make to force a draw, and thus both a win for Left and a draw are possible ways for this game to end. Thus, $G \notin \mathcal{L}$ and $G \notin \mathcal{D}$. It is clear that $G \notin \mathcal{R}$, so $G \notin \mathcal{L} \cup \mathcal{D} \cup \mathcal{R}$. ◁

We have thus seen that $\mathbb{S}$ contains games that are not part of any of the three outcome classes we have just defined. We partition the rest of $\mathbb{S}$ into a few new outcome classes: $\mathcal{LD}$ is the outcome class for games that could either end in a win for Left or a draw, in which the game G from Example 3.5 is contained. Similarly, we define $\mathcal{DR}$, $\mathcal{LDR}$ and $\mathcal{LR}$.

These outcome classes can be written compactly in a table, analogous to Table 2.1. This is done in Table 3.1. We have had to generalize the idea of winning or losing if a player starts to having a winning or drawing strategy. A player has a winning strategy if they can always force at least a win, regardless of what moves the other player makes. Similarly, a player has a drawing strategy if they can always force at least a draw, regardless of what moves the other player makes. If a player has neither, we say that the player has a losing strategy, as they can always trivially force at least a loss.

		Left has a		
		Winning strategy	Drawing strategy	Losing strategy
Right has a	Winning strategy			$\mathcal{R}$
	Drawing strategy		$\mathcal{D}$	$\mathcal{DR}$
	Losing strategy	$\mathcal{L}$	$\mathcal{LD}$	$\mathcal{LR} \cup \mathcal{LDR}$

Table 3.1: Outcome classes of synchronized games.

3.3 A Value for Synchronized Games

It would in general be useful to assign a specific value to a synchronized game, just as was done for combinatorial games. If a game $G = (\mathcal{G}^R, \mathcal{G}^L, \mathcal{G}^S) \in \mathbb{S}$ is decided, we set its value to its number value if the position was interpreted as a combinatorial game. For example, the game $(0, (), ())$ would have the value 1, mirroring the definition of $1 = \{0 \mid \}$ as a combinatorial game. It turns

out this is well-defined under some restrictions. This is also the approach taken in [3], where more details on this method can be found.

However, in general, many nice properties that hold for values of combinatorial games, do not hold for values of synchronized games. For example, we could define an addition for synchronized games by allowing players to play their move on either of the two summands, similar to how it is done for combinatorial games. As we do not use this addition after the next example we will not write out its formal definition. It turns out that in general, the value of the sum is not equal to the sum of the values of the summands.

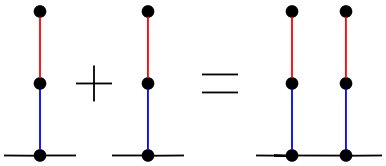


Figure 3.3: The sum of two Synchronized HACKENBUSH stalks.

Example 3.6. In Figure 3.3 one can see a sum of HACKENBUSH positions. One of the stalks on the left is a draw, as both players must play to 0, and thus has value 0. One would expect that $0+0=0$, and thus the game on the right side of the equality should also end in a draw. However, we have already seen that this game could also be a win for left, so this is not the case. ◁

Despite the value of synchronized games not respecting addition of games, we will still try to define its value for general games. It turns out that it still respects taking inverses as well as comparing games in most cases, though we will not show that in this thesis. We will do this by using tools from economic game theory, most notably Nash values. More details on this approach can also be found in [3]. Many of the economical game theory tools below are taken from [8, Section 1.2, 1.3].

Let A be a real-valued $m \times n$ matrix. We interpret this matrix as a game as follows: Left chooses a row i and Right chooses a column j of this matrix, combining to select a single element A_{ij}. Then, Right must pay an amount equal to A_{ij} to Left. If A_{ij} is negative, we interpret this to mean that Left must pay $|A_{ij}|$ to Right. We call A the *pay-off matrix* of this matrix game, and note that this game is a *zero-sum* game: any loss for one player is equal gain for the other.

We define the sets

$$X = \left\{ (x_1, x_2, \ldots, x_m)^\top \mid x_i \geq 0, \sum_i x_i = 1 \right\},$$

$$Y = \left\{ (y_1, y_2, \ldots, y_n)^\top \mid y_j \geq 0, \sum_j y_j = 1 \right\}.$$

X is the set of all *strategies* Left has, and Y the set of all *strategies* for Right. Here, x_i is the probability for Left to play on row i for $0 < i \leq m$, and, similarly, y_j is the probability for Right to play on column j for $0 < j \leq n$.

If Left has chosen a strategy $x \in X$ and Right has chosen a strategy $y \in Y$, we can calculate the expected pay-off value of A under these strategies by computing $x^\top Ay$.

Definition 3.7. A strategy $x^* \in X$ is *optimal* if ${x^*}^\top Ay' \geq \max_x \min_y x^\top Ay$ for all $y' \in Y$. Similarly a strategy $y^* \in Y$ is *optimal* if ${x'}^\top Ay^* \leq \min_y \max_x x^\top Ay$ for all $x' \in X$.

In other words, a strategy is optimal if it has the highest expected pay-off value, independent of the strategy the other player chooses. We call the expected pay-off value when both players have chosen an optimal strategy the *optimal pay-off value*.

Definition 3.8. A pair of strategies $(x^*, y^*) \in (X, Y)$ is called a *Nash equilibrium*, if for all $x \in X$ and $y \in Y$ it holds that

$$ {x^*}^\top Ay^* \geq x^\top Ay^* \text{ and } {x^*}^\top Ay^* \leq {x^*}^\top Ay. $$

In other words, if both players have a strategy and only one player changes their strategy, the resulting expected value will not be better for that player. Thus neither player has an incentive to deviate from such a strategy pair, and these are in some sense stable.

It turns out that every zero-sum matrix game has at least one Nash equilibrium, and that the expected pay-off values for all Nash equilibria are the same and are equal to the optimal pay-off value. We thus define the *Nash value* of a zero-sum matrix game to be the expected pay-off value of any of these Nash equilibria.

Using all these tools from economical game theory, we will recursively define the value of a game starting with games all of whose options are decided and build up from there. For this, we thus require the restriction we made earlier that all synchronized games have finite birthday. We already set the value of any decided $G \in \mathbb{S}$ to its number value viewed as a combinatorial game earlier. For all other synchronized games, we compute the matrix S where for all i, j we let $S_{i,j}$ be the value of $\mathcal{G}^S_{i,j}$. We then view S as a zero-sum matrix game and set the value of G to the Nash value of S. We will use the function $v \colon \mathbb{S} \to \mathbb{Q}$ as a shorthand for this definition.

Example 3.9. Let G be the HACKENBUSH position in Figure 3.2. Then $G^S = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. As G is not decided, we have that $S = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and that the Nash value of this position is $\frac{1}{2}$. Thus $v(G) = \frac{1}{2}$. ◁

Lastly, we will note a method for efficiently finding the Nash value of an arbitrary zero-sum matrix game. It turns out the Nash value of a zero-sum matrix game is equal to its expected pay-off value when both players play optimally. That is, the Nash value is equal to $\max_x \min_y x^\top Ay = \min_y \max_x x^\top Ay$. As a result, we only have to compute the value of either this maximisation or minimization problem. This can be done efficiently using Linear Programming, by solving either of the dual Linear Programs in Equation 3.9.1.

A consequence of using Nash values as the definition for values of synchronized games is that every value of a synchronized game is a fraction. For the decided positions this follows from the fact that the only numerical values a game can take are dyadic rationals, which is a subset of $\mathbb{Q}$. The value of all other positions is determined as the solution to a Linear Program with fractional coefficients, and thus the solution must also be a fraction.

There is, however, a problem that can arise when determining the value of a synchronized game in this way. If the game has positions that are decided, but one of the left or right options of that

$$\max \left\{ x_0 \;\middle|\; \begin{array}{ccc} \sum_{i=1}^{n} s_{ij} x_i & \geq x_0, & j \in \{1, \ldots, m\} \\ \sum_{i=1}^{n} x_i & = 1 & \\ x_i & \geq 0 & \end{array} \right\}$$
$$\min \left\{ y_0 \;\middle|\; \begin{array}{ccc} \sum_{j=1}^{m} s_{ij} y_j & \leq y_0, & i \in \{1, \ldots, n\} \\ \sum_{j=1}^{m} y_j & = 1 & \\ y_j & \geq 0 & \end{array} \right\}$$

Equation 3.9.1: Linear Program finding the optimal strategies of a zero-sum matrix game.

position is not decided, the ordering of two such decided positions will not necessarily match the ordering of their values. We will not go into many details here, but those can be found in [3]. We do note that this problem only occurs in Cherries and Stack Cherries games out of the five discussed in this thesis. For an example of such a position, see Figure 3.4.

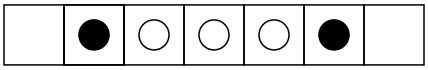


Figure 3.4: A decided Cherries position with undecided left options.

As a result, there is no clear well-defined way to define a value for Cherries and Stack Cherries games. We will go into more details on the approach we did take for these games in Section 4.2.2.

4 An Introduction to CGSynch

In the previous two sections, we have given an overview of the basics of the fields of Combinatorial and Synchronized Game Theory. One can understand that computing the canonical form of large and complicated combinatorial games by hand would be an arduous process. For this reason the application CGSuite was written [5]. It can analyse and calculate many properties of a wide range of combinatorial games, including their winner, birthday and value, among more advanced concepts such as cooling and heating of games, which we will not discuss in this thesis. However, it was only written to analyze combinatorial games and is unable to analyze synchronized games.

We have written a new `C++`-program, called CGSynch, which can determine simple properties of combinatorial and synchronized games. All of the theory discussed in Sections 2 and 3 was implemented in the program CGSynch, and is used for analyzing the rulesets described in these sections. This section will discuss how this was done, describing the general structure of the program and noting where different theorems were used in the program. Further on we will describe the interface of CGSynch. The program can be downloaded from our GitHub repository, where the source code is also available [10].

The program CGSynch can read in positions of five different games: Cherries, Hackenbush, Stack Cherries, Push and Shove. It can compute multiple properties of these positions, such as their value, outcome class and birthday in both the combinatorial and synchronized sense. Furthermore, for combinatorial games, it can add and subtract them, compare them, put them in canonical form and simplify their value to a number, if it is one.

4.1 The Trees

The main structure of the program revolves around two types of trees: abstract trees and game trees. Each node of a game tree represents a unique position of a game, while each node of an abstract tree represents the mathematically abstracted version of that position. The structures are very similar for combinatorial and synchronized games. In this section, we will first explain how they work for combinatorial games, and then note the few differences with synchronized games.

We will first focus on the game tree. The game tree is the same as it is in the combinatorial sense. It has a single position of the ruleset as the root of the tree, and the children of each position are the positions that can be reached by either Left or Right making a single move. An example of such a game tree can be seen in Figure 4.1.

For every ruleset, we have written a class that contains a position of that ruleset and can determine what positions can be reached by Left and Right from this position. Instances of these classes are used as the nodes in a game tree. Each of the positions of such an instance and all its transpositions are saved in a database. Whenever a new node is added to the game tree, the database is first checked whether this position has occurred before. If this is the case, we instead use this position saved in the database, as we have already determined its children, reducing the time spent building up the game tree.

The abstract tree is similar, but instead of each node in the tree representing the position in a game, it represents a mathematical abstraction of such a position. A node of the abstract tree relates to a node in the game tree in the same way that $\{-1 \mid 1\}$ relates to the root of the game tree in Figure 4.1.

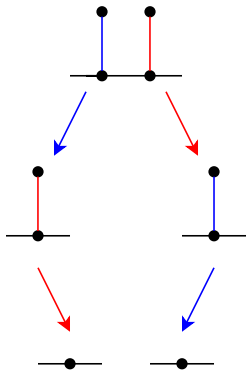


Figure 4.1: An example of a game tree.

The abstract tree is defined by its left and right option sets. As a result, instead of it being generated down like the game tree, it is built upwards. Starting at the leaves of the abstract tree (all of which are empty positions), we can generate new nodes by defining their left and right option sets as previously defined nodes. In this fashion, any combinatorial game can be reached. We again use a database for storing every abstract node and making sure not to have any duplicates. We also cache every property of the game corresponding to an abstract node when it is calculated in that abstract node. As a result, we never have to determine a mathematical property of the same game multiple times, even if it appears multiple times in the abstract game tree.

When analyzing a position, such as the one in Figure 4.1, we first generate the entire game tree. Then, starting from the 0-positions at the bottom of the game tree, we construct the leaves of the abstract tree. From there we build up the abstract tree layer by layer. Each time we save an ID of an abstract tree node in the corresponding game tree node, and then use these IDs to build up the next layer of the tree. We repeat this until the entire abstract tree is built. When analyzing the position of this ruleset with the combinatorial tools described in Section 2, we analyze the abstract tree. An example of both trees for a simple HACKENBUSH position can be found in Figure 4.2.

For synchronized games, we work very similarly. However, as synchronized positions are not only defined by their left and right options but also by a matrix with all synchronized options, there are some minor differences. When generating the game tree, we do not only generate two sets of positions, but also a matrix of positions reached after both players have made a move.

When determining the abstract tree that corresponds to a certain game tree, we again create a matrix of IDs of tree nodes as well as two sets for the left and right options. The abstract database is then queried to find a corresponding abstract node. However, we only check if there is a game with an equal matrix in the database, not one with an equal matrix under permutation. Determining whether this is the case often costs quite some time, as permutation equality with every single matrix already in the database needs to be checked. Due to graphs being able to be represented by matrices by writing them as an adjacency matrix, this problem is at least as hard as solving graph isomorphism, for which no known polynomial algorithm exists at the time of writing [6].

In all other regards, the algorithm works the same as the combinatorial form.

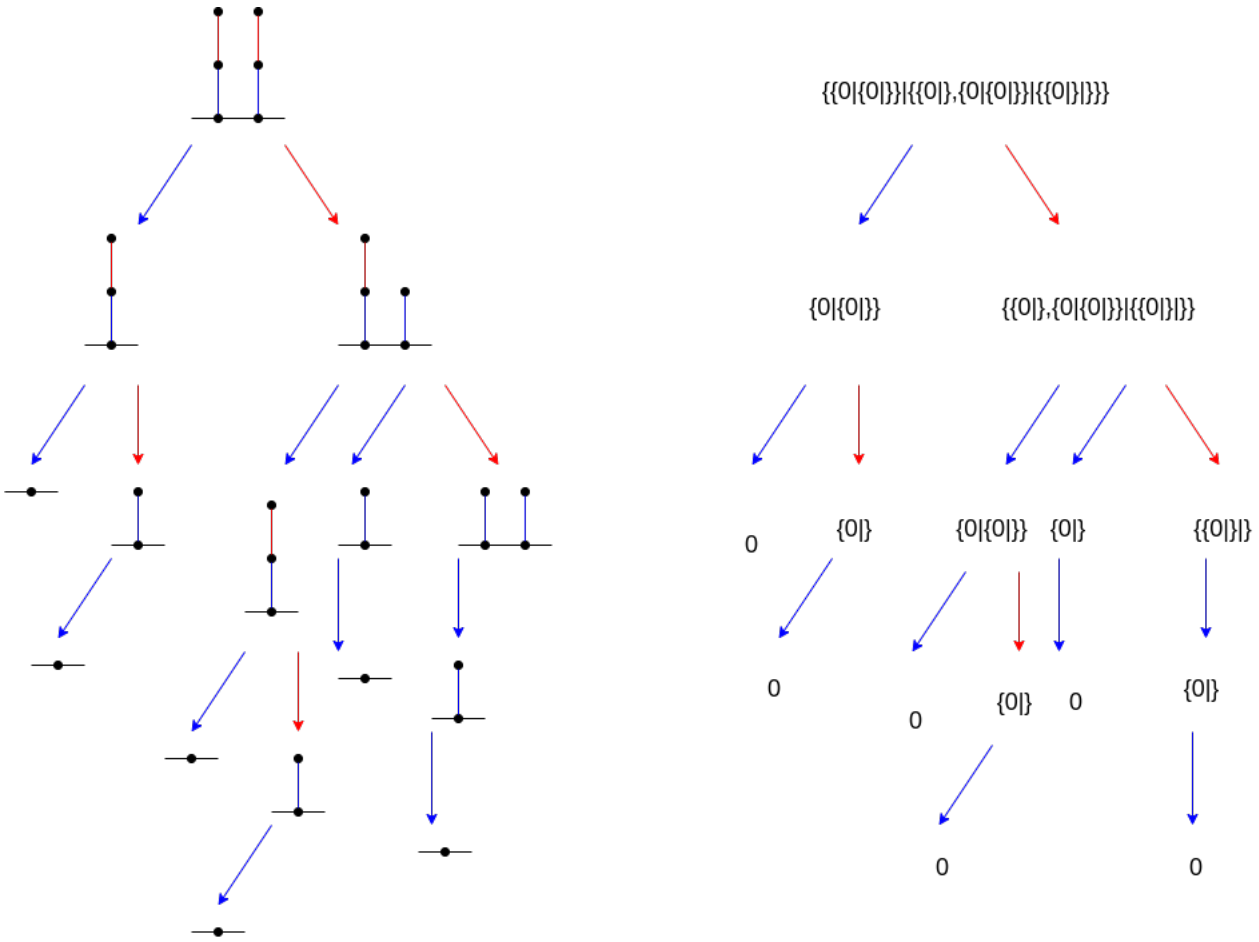


Figure 4.2: An example of the game tree and abstract tree of the same HACKENBUSH position. Note that due to usage of a database in reality equivalent nodes are merged in both trees.

4.2 How Analysis of Games is Performed

This section will explain exactly what theorems are used to compute different properties of combinatorial and synchronized games.

4.2.1 Analyzing Combinatorial Games

A few basic properties of positions are determined recursively from the definition. This applies to determining the winner of a position, using Table 2.2, and the birthday of a position, using Definition 2.4. Adding two games is done recursively using Definition 2.7. Taking inverses is done using Definition 2.9. Subtracting one game from another is also basically done from the definition, but we combine the addition with taking the inverse. For example, for two games $G, H \in \mathbb{G}$ we compute $G - H$ recursively as $\{\mathcal{G}^L - H, G - \mathcal{H}^R \mid \mathcal{G}^R - H, G - \mathcal{H}^L\}$.

We compare two games, say $G \sim H$ with $\sim \in \{<, \leq, \lhd\!\mid, >, \geq, \mid\!\rhd, \neq, =, \|\}$, in two steps. First, we rewrite this as $(G - H) \sim 0$. Secondly, we calculate the winner of the game $G - H$ and use Theorems 2.16 and 2.18 to determine whether this comparison holds.

We use a few tricks to speed up this comparison. When the canonical form of either G or H has already been calculated earlier, we use those for the subtraction instead of G and H directly. These have smaller game trees, so calculating their difference is faster. Additionally, if H has a smaller birthday than G, we instead calculate $H - G$ and reverse r to compute the comparison. This saves time, as the inverses of all positions in the game tree of the right operand of the subtraction need to be calculated, and having a smaller birthday often indicates having a smaller game tree. Lastly, if both G and H are known to be numbers, we calculate their numerical value and compare those.

When we convert a game G to its canonical form, we take the following four steps. First, we convert every direct option of G to its canonical form. This simplifies comparisons between these two options later. Secondly, we remove all dominated options, and all but one copy of every equal option. Thirdly, we check if the Simplest Number Theorem 2.28 applies. Normally this applies if all left and right options are numbers and all left options are smaller than all right options. However, as we already have removed all dominated options from G, this would now imply that both Left and Right only have a single option left, both a number, such that the left option is smaller than the right option. By Theorem 2.29 we then have that G equals the simplest number between these two numbers. Lastly, we check if any of our options is reversible. If any of them are, we add the new left and right options to G's option sets, and recursively restart the algorithm on the newly created game. This is, as adding those new options may have added options that could be dominating or be dominated by already known options of G, resulting in more options that have to be removed.

Lastly, we check if a game G is a number as follows. First, we convert it to its canonical form if it is not in canonical form yet. Next, we check if it is an integer, simply by checking that it corresponds to Definition 2.23. If it is not an integer, it can only be a number if it is a dyadic rational. We thus check that it has exactly one left and right option, that both of these are numbers, and that the left option is smaller than the right option. If all of this is true, then by the Simplest Number Theorem this must be a number as well.

4.2.2 Analyzing Synchronized Games

Compared to analyzing combinatorial games, CGSynch can compute relatively few properties of synchronized games. It can, however, calculate their birthday, using Definition 3.3. Additionally,

it can compute the outcome class of a game, which is done recursively using the definition in Section 3.2. Each of these properties is cached so that we do not have to compute the same result twice.

Lastly, CGSynch can calculate for any synchronized game G its value $v(G)$. It does this recursively. It determines the value of $\mathcal{G}_{ij}^{S}$ for all i, j, and puts all of these in a matrix S. This matrix is then converted to the Linear Program in Equation 3.9.1, which is solved using the application Gurobi [7]. This procedure is done recursively until a decided game is reached. For every different ruleset, a function is implemented that can compute the value of a decided position from that ruleset, which is then called to determine the value.

As the Synchronized Cherries and Synchronized Stack Cherries contain decided positions with undecided options, determining their value is, in general, not well-defined. Whenever someone tries to do so anyway, we show a warning but continue by instead using the following metric. Note that this will in general not give correct results, but it will still give a result.

If a Synchronized Cherries or Synchronized Stack Cherries position is decided and it has no undecided options, all of its stones must be of the same colour. In that case, if the stones are black, we set the value of the game equal to the number of stones in the position, and minus the number of stones if they are white. This is similar to how Hackenbush positions with only one colour of edge have value equal to the number of edges, negated if all edges are red. As we've seen in Figure 3.4 it is possible for a Cherries or Stack Cherries position to be decided but have undecided options. In all such positions, there are still stones of both colours left, but only one of the players is able to move. In this case, we approximate its value by taking the number of moves that that player still has, again negating if that player is White. This is not a perfect or mathematically sound definition but does give a value that corresponds roughly with its actual value.

4.3 The User Interface

CGSynch has a different interface for entering combinatorial and synchronized games. This is a result of different operations being implemented on both of them. We will describe the interfaces for combinatorial and synchronized games in a different subsection each. An example of how the user interface looks can be found in Figure 4.3.

In both cases, the user interface is implemented as a command-line interface, where one can create games using a position, and then ask for properties of these. For example, one can create a Cherries game with a given position and determine its birthday using the command `Cherries(POSITION_STRING).GetBirthday()`. When interpreted as `C++`-code, this looks like an instance of the class `Cherries` is created with the given position, and afterwards the function `GetBirthday()` is called on this instance. This is indeed how it is meant to be read. All implemented properties of games can be queried using this same "initialize class and call property function" template.

Every command entered in the combinatorial interface must conform to the grammar given in Appendix A.

When creating an object of type `Shove`, `Push` or some other game, a string needs to be passed representing the position of that game. As Shove, Push, Cherries and Stack Cherries are

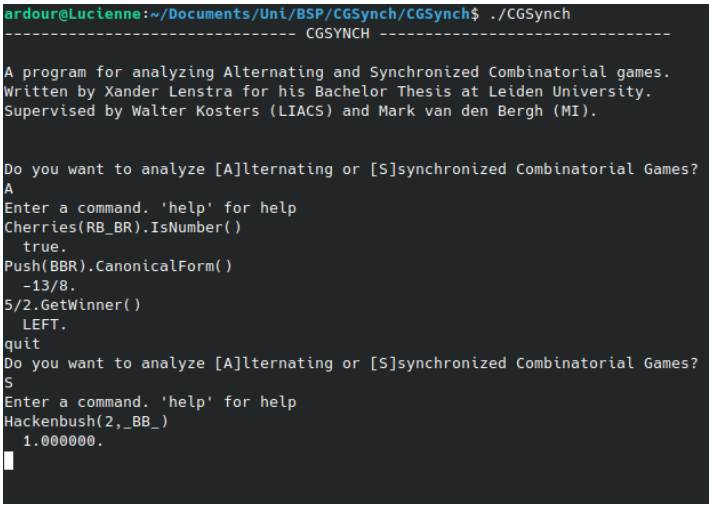


Figure 4.3: Example of the interface of CGSynch.

played on a strip, the input string will be read as a strip, where each 'R' or 'W' will be turned into a red (or white) piece, each 'B' will be turned into a blue (or black) piece, and each underscore and space will be turned into an empty square. If any other character is encountered, an error will be printed.

Example 4.1. The Shove position in Figure 2.5 can be created in CGSynch with the input `Shove(BR_R__B)`. ◁

Creating a Hackenbush game is more complex, as that game is played on a graph instead of a strip. To input this position, we enter the adjacency matrix of this graph. The first row and column of the adjacency matrix must correspond to the "ground" node of the position. The same letters as before are used for the different colours of edges. All rows of this matrix are concatenated into a single string so it can be entered as a single line. The size of the adjacency matrix has to be passed as an additional argument to aid in reconstructing the matrix, and to check that the passed string has the right length.

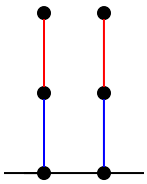


Figure 4.4: A simple Hackenbush position.

Example 4.2. The adjacency matrix of the Hackenbush position in Figure 4.4 is

$$\begin{pmatrix} \_ & B & B & \_ & \_ \\ B & \_ & \_ & R & \_ \\ B & \_ & \_ & \_ & R \\ \_ & R & \_ & \_ & \_ \\ \_ & \_ & R & \_ & \_ \end{pmatrix}.$$

This is concatenated to a single string, `_BB__B__R_B___R_R____R__`. As the adjacency matrix is a 5×5 matrix, the expected input for CGSynch to generate this Hackenbush position is: `Hackenbush(5,_BB__B__R_B___R_R____R__)`. ◁

When an object of type `Shove`, `Push` or some other game is turned into an `AbstractGame`, we explore the entire game tree starting from that position, and then build the abstract tree bottom-up, as was explained in Section 4.1.

Example 4.3. Below are a few more example inputs and their outputs.

- `Shove(BRRB).GetWinner()` results in `LEFT`.
- `(Push(RBR) - Shove(R)).CanonicalForm()` reults in `-7/8`.
- `Push(R).Birthday()` results in `1`.
- `Hackenbush(3,_BRB__R__) == Hackenbush(2,_BB_) + Hackenbush(2,_RR_)` results in `True`.
- `3/4 + 3/4` results in `3/2`.
- `Cherries(B__R_B) >= 3` results in `True`.
- `(-6/8).GetWinner() >= 5.GetWinner()` results in `False`.
- `{3|4}.CanonicalForm()` results in `7/2`. ◁

One can find a video with a demo of a few of the possibilities of analyzing a Hackenbush position using CGSynch in [9].

Due to there being less functionality implemented for synchronous games, the grammar for the synchronized UI is smaller. It can be found in Appendix B.

As one can see this is effectively a subset of the grammar for the combinatorial interface, where integers, fractions, addition and subtraction of games, comparisons between games and comparisons between winners have been removed. As such, we will not go into detail on how exactly inputs need to be structured or give examples. All of these can be found in the description of the combinatorial interface.

5 Results and Limitations

In this section, we will first discuss experiments done with CGSynch and their results. These experiments will show the speed and accuracy of CGSynch. Afterwards, we will note a few things that were not implemented in CGSynch.

All of these experiments were run on a Linux desktop with 48 GB RAM and a Six-Core Processor, overclocked to 4142.051 MHz. The program was compiled with `g++` 9.4.0, with the flag `-O3` enabled.

5.1 Accuracy of CGSynch

We have performed multiple experiments testing the accuracy of CGSynch. We first analyzed combinatorial positions of Push in both CGSynch and CGSuite and compared those. For this, experiment, we generated 10 random Push positions on a strip of length 14, and asked both CGSynch and CGSuite to determine the canonical form of this position. The results of this experiment can be seen in Table 5.1. Both CGSynch and CGSuite find the same values in all of the 10 Push positions, which highly suggests that CGSynch can correctly determine the canonical form of simple Push positions.

Input	CGSynch & CGSuite
`Push(___R_R__B___BR).CanonicalForm()`	$-27983/16384$
`Push(BR_BRBRRB_R_BB).CanonicalForm()`	$2318031/131072$
`Push(_B_B_B_BRBRBRR).CanonicalForm()`	$-20513867/1048576$
`Push(_RRB_B_B__RBR_).CanonicalForm()`	$-119283/32768$
`Push(RRBR_BBRRBRBRR).CanonicalForm()`	$-12216755/524288$
`Push(RRBR_BBRBR_R__).CanonicalForm()`	$-512947/32768$
`Push(B__RBRRB_____B).CanonicalForm()`	$513969/32768$
`Push(BB_B_BBBRRB___).CanonicalForm()`	$7341858679/4294967296$
`Push(RR_R_BRBB_BR_R).CanonicalForm()`	$-8398759211/536870912$

Table 5.1: Determining the canonical form of simple Push positions using both CGSynch and CGSuite.

Next, we have investigated the analysis of Cherries positions. As CGSuite is unable to analyze positions of this game, we will instead compare values obtained using the formula from Section 2.7 and those obtained by CGSynch with this optimization turned off. For this, we ask CGSynch to compute the canonical form of 10 different Cherries positions. The results of this can be seen in Table 5.2. Again, the results from CGSynch line up exactly with the theoretical results.

Based on these two tests, it seems likely that CGSynch can indeed correctly determine the canonical form of different combinatorial games. As determining the canonical form of a position also requires other properties to be determined correctly, such as comparing games, subtracting games and determining their winner, this also suggests that all those properties are working as intended.

Lastly, we have investigated whether CGSynch can correctly compute the value of a few positions of Synchronized Hackenbush to test its accuracy for synchronized games. We define H to be the Synchronized Hackenbush position consisting of a red edge on top of a blue edge and

Input	CGSynch & Theory
`Cherries(RRRRRRRR__BBRB_B_R__BR__RBBBB).CanonicalForm()`	-3
`Cherries(_BBRBRR__BBRBR_RB_BRBBB___R__).CanonicalForm()`	3
`Cherries(B_RRBBBBB_B_RR_B_R__RR_BBRRBR).CanonicalForm()`	1
`Cherries(R_RBRRB__RRBBBRRB_BRRB__BRB__).CanonicalForm()`	-2
`Cherries(_RBB__BR__R_R_BRB__BRR__BB_RR).CanonicalForm()`	-1
`Cherries(R__R_R_BB_B_RR__RRRRR_BRRRBR_).CanonicalForm()`	-8
`Cherries(__BR__RB_BB_R_RR_BB___BBRB_RR).CanonicalForm()`	1
`Cherries(_B_B_B__RRBBR___RB__B_RRBR__B).CanonicalForm()`	2
`Cherries(RBRBR_RR____RRBBRRR_BR_BRRR_R).CanonicalForm()`	-9

Table 5.2: Determining the canonical form of simple Cherries positions using both CGSynch and CGSuite.

$-H$ to be the position of a blue edge on top of a red edge. We then set nH to the Hackenbush position consisting of n copies of H, and similarly we set $-nH$ to the position consisting of n copies of $-H$ for all $n \in \mathbb{N}$. An example of such a position can be seen in Figure 5.1.

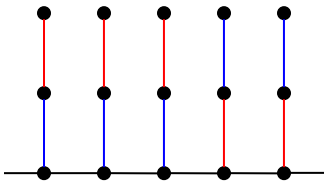


Figure 5.1: An example of the Hackenbush stalks position $3H - 2H$.

These Synchronized Hackenbush positions were analyzed before, and a table of values was computed for all positions $nH - mH$, with $n, m \in \mathbb{N}_{\leq 10}$ by Mark van den Bergh [2]. We have computed this table for $n, m \leq 5$ using CGSynch, and the result can be seen in Table 5.3. The values computed by CGSynch are again exactly the same as those computed previously, and thus it is likely CGSynch can correctly compute the values of synchronized games.

$m \backslash n$	0	1	2	3	4	5
0	0	0	0.5	0.833333	1.3333333	1.733333
1	0	0	0.25	0.642857	1.006485	1.458537
2	-0.5	-0.25	0	0.25	0.666189	1.058927
3	-0.833333	-0.642857	-0.25	0	0.25	0.666189
4	-1.333333	-1.006485	-0.666189	-0.25	0	0.25
5	-1.733333	-1.458537	-1.058927	-0.666189	-0.25	0

Table 5.3: Table of values of the Synchronized Hackenbush positions $nH - mH$.

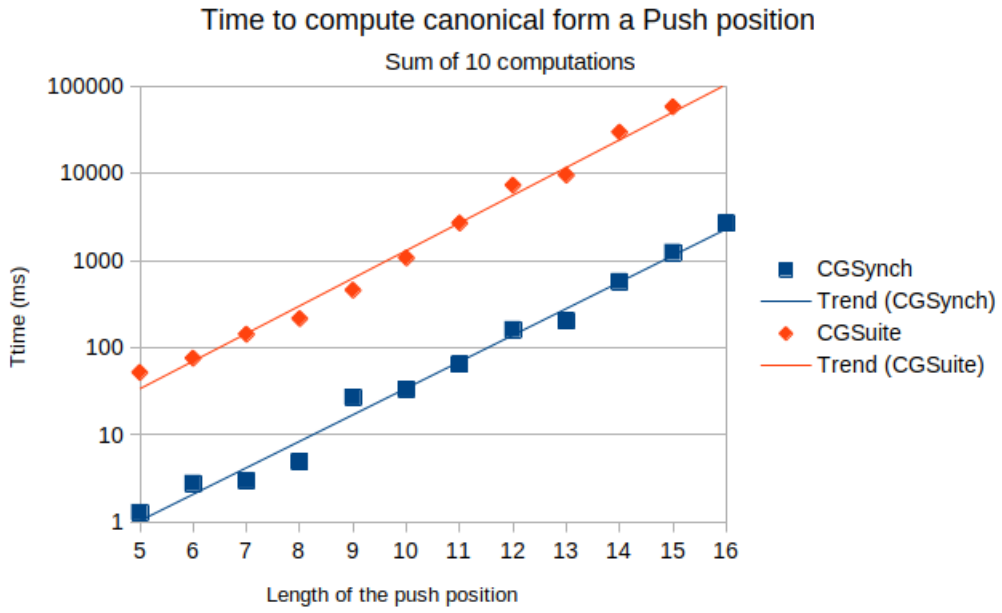


Figure 5.2: Time it takes to compute the canonical form a PUSH position.

5.2 Speed of CGSynch

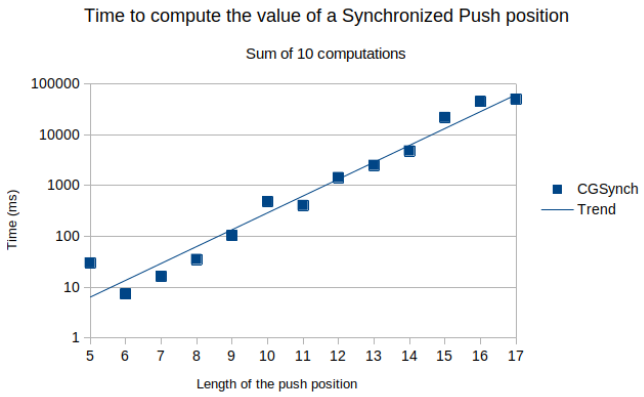
Even if a program is accurate, having a result quickly is also important. We have thus generated 10 random PUSH positions with strips ranging from length 5 to 16, and ask both CGSUITE and CGSYNCH to compute their canonical form. We run all tests in order to also test the caching used by both CGSYNCH and CGSUITE. The results can be seen in Figure 5.2. The horizontal axis represents the length of the PUSH positions, and the vertical axis represents the total time to analyze all 10 positions. The data point for CGSUITE at length 16 was not included, as in at least one case CGSUITE crashed while preforming the calculation. It is clear that CGSYNCH is almost 30 times faster than CGSUITE on the provided test set.

For synchronized games, we have run a similar test to determine its speed. However, as CGSUITE cannot analyze these games, we cannot compare our results against its values. To compensate for this, we have analyzed the speed of both SYNCHRONIZED PUSH and SYNCHRONIZED HACKENBUSH.

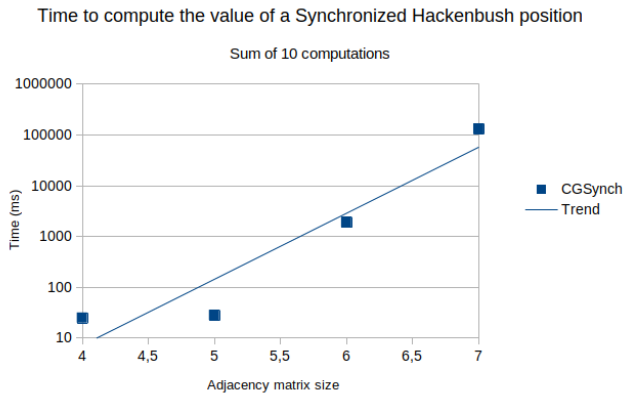
For SYNCHRONIZED PUSH, we have generated 10 random strips for each of the lengths from 5 up to 17 and let CGSYNCH compute the value of those positions. The results of this can be found in Figure 5.3a. The very first data point is a bit of an outlier, which is caused by the very first game. Determining its value took 20.0 ms while determining the value of all other 9 positions of length 5 only took 10.0 milliseconds in total. This is likely due to many of the smallest games first being generated when computing this position, which other games can then simply reuse.

For SYNCHRONIZED HACKENBUSH we have again generated 10 random graphs and adjacency matrices of size 4 to 7. Each edge of each of these graphs was randomly coloured or removed with probability $\frac{1}{3}$. CGSYNCH was asked to compute the value of these positions. The results can be

found in Figure 5.3b. Again, the very first data point is an outlier, as the first game of the set of ten took 23.2 ms of the 24.8 ms that the entire set took. As in the previous experiments, there is a clear exponential increase in the time used as the position becomes more complicated. However, the slope of the exponential curve is much steeper than that of PUSH. In PUSH each increment in strip length only increases the number of moves of either player by at most 1. However, in HACKENBUSH an increment of the size of the adjacency matrix from n to $n+1$ allows for $n+1$ more edges, and so at most $n+1$ more moves. It is thus expected that this exponential curve has a steeper slope.



(a) Time for computing the value of a SYNCHRONIZED PUSH position.



(b) Time for computing the value of a SYNCHRONIZED HACKENBUSH position.

Figure 5.3: Speed of computing the value of games from two different Synchronized rulesets.

One might ask why CGSYNCH is so much faster than CGSUITE. The programming language in which both are written at least partly explains this. CGSUITE is written in Scala and compiled to Java bytecode, which then has to be interpreted when running. CGSYNCH is written in `C++`, which can be compiled directly to native bytecode, which runs much faster. Another factor is the aggressiveness with which CGSYNCH caches results. CGSYNCH caches the result of almost every single calculation. This has the obvious advantage that no result has to be calculated twice, thus saving time in the long run. However, this does increase the amount of memory CGSYNCH uses. While running the previous tests, CGSYNCH sometimes used up to 20GB of memory.

5.3 A Conjecture on the Value of Synchronized Shove

When testing CGSYNCH on SYNCHRONIZED SHOVE positions, we came across an interesting result. All of the positions consisting of a single strip appeared to have integer values. We have checked that this holds for all approximately ~ 4.7 million SHOVE positions of length 13 or less, but do not have a general proof for this.

Notably, the same is not true for SYNCHRONIZED PUSH. The smallest counterexample is the position `Push(BR_BR)`, which has -2.5 as its value, as its corresponding zero-sum pay-off matrix is $\begin{pmatrix} -2 & -3 \\ -3 & -2 \end{pmatrix}$.

5.4 Limitations

Even though many features are implemented in CGSynch, there are still a few notable features missing that one might expect. These were mostly left out due to a lack of time. This section will go over a few of these features. Additionally, some optimizations that were not implemented will be mentioned.

CGSynch uses quite some memory as it caches every single result that has ever been computed. Implementing a system that caches only a certain number of games based on how recent they have been used in other computations is certainly possible and would add a simple maximum to the amount of memory used. It would also, however, in some cases reduce the speed of CGSynch as some results would sometimes need to be recalculated.

Furthermore, not many properties of Synchronized Games can be computed. For example, CGSynch cannot determine if a game is decided but contains undecided positions. It also cannot add or compare two Synchronized Games. All of these can of course be implemented, and, in our opinion, this should be easily doable, but the current version of CGSynch does not support them. It is also currently not possible to input Synchronized Shove and Synchronized Push positions consisting of more than one strip, which is often desirable.

Lastly, the interface of CGSynch is very basic. Many quality of life changes could still be implemented, such as being able to scroll back through previously entered queries, having some sort of auto-completion or showing understandable error messages when incorrect input is entered. Alternatively, having an actual interface for CGSynch, instead of just a command line, would also be an improvement.

6 Conclusion

In this thesis, we have discussed the basics of Combinatorial Game Theory and Synchronized Game Theory. We have created a program, CGSynch, that can determine basic properties of games from these rulesets. We have explained how this program works, and what theorems and definitions are used to compute properties of games.

Furthermore, we have done experiments to verify the accuracy and demonstrated the speed of CGSynch. It became clear that CGSynch is able to accurately determine the canonical form of combinatorial games, as well as the value of synchronized games. As calculating these values requires many other properties also to be determined correctly, we also concluded that comparison, subtraction and determining the winner of a position are likely determined correctly. From our speed tests we also concluded that CGSynch was faster than CGSuite in computing the canonical form of combinatorial games. It did, however, also use much more memory. For synchronized games we also showed that CGSynch was able to compute the value of many games within a reasonable amount of time, though exactly how long increased exponentially based on the size of the position.

All in all, we have created an application that could be used for analyzing combinatorial and synchronized games and is faster than current alternatives.

There is, of course, still room for future work. While this program can determine simple properties, many properties of synchronized games cannot be analyzed, as we already noted in Section 5.4. For example, implementing features such as addition of synchronized games or determining if a synchronized position is decided or terminal can be added in the future. Similarly, we can only determine basic properties of combinatorial games. CGSynch could also be extended to deal with impartial games, cooling and heating games or determining the atomic weight of a position. All of these things can be looked at in future work.

Alternatively, the features CGSynch already has can be used in future work for analyzing synchronized games.

References

[1] Michael H. Albert, Richard J. Nowakowski, and David Wolfe. *Lessons in Play*. 2nd ed. CRC Press, 2019.

[2] Mark van den Bergh. Private Communication. 2022.

[3] Mark van den Bergh. "Combinatorial games and imperfect information variants". Unpublished.

[4] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*. 2nd ed. Academic Press, 1982.

[5] *CGSuite*. URL: http://cgsuite.sourceforge.net.

[6] Martin Grohe and Pascal Schweitzer. "The graph isomorphism problem". In: *Communications of the ACM* 63.11 (2020), pp. 128–134.

[7] *Gurobi*. URL: Gurobi.com.

[8] L.C.M. Kallenberg, F.M. Spieksma, and M.J.H. van den Bergh. *Discrete Besliskunde*. 2019. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxkYmtsZWlkZW58Z3g6MWQwYjc5MDU5ODdhZjVmNw.

[9] Xander Lenstra. *CGSynch Demo*. URL: https://youtu.be/X_NWDsAo8Ns.

[10] Xander Lenstra. *Code for CGSynch*. URL: https://github.com/xlenstra/CGSynch.

[11] Thomas de Mol. "Synchronized Cherries". Bachelor Thesis. Universiteit Leiden, 2021. URL: https://theses.liacs.nl/pdf/2020-2021-MolTde.pdf.

[12] Aaron N. Siegel. *Combinatorial Game Theory*. American Mathematical Society, 2013.

[13] Zeycus. *File:Hackenbush Girl.svg*. Used unaltered, licensed under GNU Free Documentation Licence v. 1.2. URL: https://en.wikipedia.org/wiki/File:Hackenbush_girl.svg.

Appendices

A Grammar of the Alternating Interface

In this grammar, we use text between square brackets [] to denote a description written out in English. String literals are written between backticks (`). A star $*$ is the Kleene star, used to denote that the preceding part of the input may be repeated any number of times, or not appear at all.

Due to there being left recursion in the definition of `AbstractGame`, the definition of the grammar used internally in CGSynch is slightly different. However, the accepted inputs are the same as in the one below.

```
Integer        = [any number]*
QuotedString   = `"` [any character except "]* `"`
String         = [any lower or uppercase letter, number, _ or space]*
                 | QuotedString
Fraction       = Integer `/` Integer
Shove          = `Shove(` String `)`
Push           = `Push(` String `)`
Cherries       = `Cherries(` String `)`
StackCherries  = `StackCherries(` String `)`
Hackenbush     = `Hackenbush(` Integer `,` String `)`

AbstractGame   = Shove
                 | Push
                 | Cherries
                 | StackCherries
                 | Hackenbush
                 | Integer
                 | Fraction
                 | `(` AbstractGame `)`
                 | `-` AbstractGame
                 | AbstractGame `+` AbstractGame
                 | AbstractGame `-` AbstractGame
                 | AbstractGame `.CanonicalForm()`
                 | `{` AbstractSet `|` AbstractSet `}`
AbstractSet    = AbstractGame (`,` AbstractGame)*
                 | ε

Winner         = AbstractGame `.GetWinner()`
Boolean        = AbstractGame `>` AbstractGame
                 | AbstractGame `>=` AbstractGame
                 | AbstractGame `==` AbstractGame
                 | AbstractGame `=` AbstractGame
                 | AbstractGame `<=` AbstractGame
                 | AbstractGame `<` AbstractGame
```

```
        | AbstractGame '||' AbstractGame
        | AbstractGame '<|' AbstractGame
        | AbstractGame '|>' AbstractGame
        | AbstractGame '!=' AbstractGame
        | string '<' String
        | String '==' String
        | String '=' String
        | String '>' String
        | String '!=' String
        | Winner '<' Winner
        | Winner '<=' Winner
        | Winner '==' Winner
        | Winner '=' Winner
        | Winner '>= Winner
        | Winner '>' Winner
        | Winner '!=' Winner
        | AbstractGame '.IsNumber()'
        | AbstractGame '.IsInCanonicalForm()'

Output  = Boolean
        | Winner
        | AbstractGame '.GetBirthday()'
        | AbstractGame '.DisplayString()'
        | AbstractGame
        | String
```

B Grammar of the Synchronized Interface

```
Integer        = [any number]*
QuotedString   = " [any character except "]* "
String         = [any lower or uppercase letter, number, _ or space]*
                 | quotedString
Shove          = `Shove(` String `)`
Push           = `Push(` String `)`
Cherries       = `Cherries(` String `)`
StackCherries  = `StackCherries(` String `)`
Hackenbush     = `Hackenbush(` Integer `,` String `)`
AbstractGame   = Shove
                 | Push
                 | Cherries
                 | StackCherries
                 | Hackenbush
                 | `(` AbstractGame `)`

WinnersSet     = AbstractGame `.GetWinners()`
Boolean        = AbstractGame `==` AbstractGame
                 | AbstractGame `=` AbstractGame
                 | AbstractGame `!=` AbstractGame
                 | WinnersSet `==` WinnersSet
                 | WinnersSet `=` WinnersSet
                 | WinnersSet `!=` WinnersSet
                 | String `==` String
                 | String `=` String
                 | String `!=` String

Output         = Boolean
                 | WinnersSet
                 | AbstractGame `.GetBirthday()`
                 | AbstractGame
                 | String
```