#A13 INTEGERS 21B (2021)

GENETICALLY MODIFIED GAMES

Melissa A. Huggan[1]
Department of Mathematics, Ryerson University, Toronto, Ontario, Canada
melissa.huggan@ryerson.ca

Craig Tennenhouse
School of Mathematical and Physical Sciences, University of New England, Biddeford, Maine
ctennenhouse@une.edu

Received: 1/15/21, Revised: 7/2/21, Accepted: 10/5/21, Published: 12/20/21

Abstract

Genetic programming is the practice of evolving formulas using crossover and mutation of genes representing functional operations. Motivated by genetic evolution we introduce and solve two combinatorial games, and we demonstrate some advantages and pitfalls of using genetic programming to investigate Grundy values. We conclude by investigating a combinatorial game whose ruleset and starting positions are inspired by genetic structures.

1. Introduction

The fundamental unit of biological evolution is a gene, which represents a small piece of information, and the genome is a collection of genes that encodes an organism's complete genetic information. Within the context of biological evolution, the genes of the most fit organisms survive and are passed on to the next generation, with their chromosomes modifying over time to better fit their environment through competition. This modification occurs through the processes of mutation and crossover, wherein individual genes are altered and pairs of chromosomes trade information, respectively, as organisms pass down their genetic information to their progeny (see Figure 1).

This set of mechanisms in biological evolution has been co-opted as a model for algorithmic development of heuristic solutions to a variety of problems, like antenna design [14], the Traveling Salesman Problem [7], and graph coloring [10]. In these problems a chromosome encodes information about the structure and properties of

[1]Supported by the Natural Sciences and Engineering Research Council of Canada (funding reference number PDF-532564-2019).

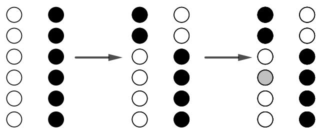

Figure 1: A pair of chromosomes undergoing crossover then mutation. In general, as exemplified by a grey gene, this process is not restricted to two values.

working solutions. These solutions are the results of *genetic algorithms*. When the chromosome instead represents a function or program the process is called *genetic programming*. Genetic programming is often used when a user has a collection of data points and is looking for a function to fit them. The fitness of a particular program is therefore related to the error between the data points and the program. This mechanism is similar to that of regression in statistical methods (Figure 2). Given a set of data points, both tools are used to minimize the error between these data points and the associated function values. Statistical regression requires a predefined model in which the coefficients are optimized. In genetic programming, the function itself is iteratively adjusted, via specified operations called crossover and mutation, using a pool of elementary functions and constants. No predefined model is necessary. If the resulting function better fits the data, the change is likely to be accepted and the process continues. The process stops evolving based on a predetermined number of iterations or when a particular fitness threshold is reached.

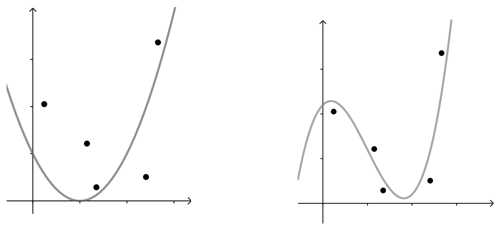

Figure 2: A curve with poor fit to data points (left) and another with much better fit (right).

There are a number of different structures used in genetic programming to represent a chromosome, the simplest being a linear structure and a tree structure. A linearly organized chromosome can be visualized much like a biological chromosome (see Figure 1). One example of such a structure is in [16], which introduces Multi-Expression Programming. Chromosomes represented by trees have some advantages over the linear approach, and we will discuss them further in Section 1.1. As the structures representing a chromosome undergo crossover and mutation, the functions associated with them exhibit smaller errors and better fit the goal data.

Genetic algorithms have been applied to combinatorial games, (see [6, 13, 18]). However, these efforts have been focused on using genetic programming to develop strategies rather than finding a formula for the Grundy values. We are interested in examining whether genetic programming could be a useful tool for determining values of combinatorial game positions. The only model for this type of project in the literature is in [16], which uses the Multi-Expression Programming model with a linear chromosome. This method does not precisely fit our needs, as the author of that paper focuses on the outcome class classification problem instead of Grundy values, and restricts their investigation to NIM. However, the project and its success serve as a strong motivator for the application of genetic programming to combinatorial games, and we hope we have done it justice in extending their results and adding to the body of work combining these two mathematical endeavors. An automated conjecturing tool was examined in [6] within the context of the game CHOMP, also focused on outcome classes.

Recall that the *Grundy value* of an impartial game position is the smallest non-negative integer not included in the Grundy values of its options [19, 11]. For more information about combinatorial game theory, see [1, 2, 3, 4, 5, 8]. In this project we generate data points of the form (H, g), where $H \in \mathbb{Z}^n$ is a list of integers representing a game position, and $g \in \mathbb{Z}$ is the associated Grundy value. A key challenge of this project is the fact that heuristics are not often useful for calculating Grundy values. In truth, either a function completely determines the value of a game or it is incorrect. This leaves us with the difficult task of devising a fitness function that represents distance, not a natural concept in the space of impartial game values, and at the same time leads to eventual convergence with an error of zero.

The paper proceeds as follows. We first give necessary background for genetic programming, framed within the context of the Python package `gpLearn` and the methodology for the conducted experiments. We then introduce, in sequence, three impartial rulesets motivated by genetic structures. The first two games use simplified structures to allow for computational analysis. In Section 2, we utilize `gpLearn` as a tool to help conjecture a pattern within the sequence of game values. This then points us in the direction of the known game KAYLES, to which we prove its equivalence. Next, in Section 3, we introduce a two-point crossover game and again implement our computational methods to obtain conjectured formulas for Grundy values based on the number of heaps. Although this alone does not prove the generalized formula, the program provides a direction for the structure of a general formula which we prove in Theorem 3.2. Lastly, we examine a game whose ruleset is inspired by more typical genetic changes, and includes both crossover and mutation moves. In Theorem 4.3 we prove its equivalence to a subset of ARC KAYLES positions and in Theorem 4.5 we prove their game values. We conclude with future directions.

1.1. Genetic Programming: Methods

Since we are interested in data points that are computationally inexpensive to determine, we have chosen to use the Python package gpLearn [20]. This package uses the tree model of chromosome representation, introduced in the introduction to Section 1, wherein each leaf is associated with a primitive (a constant or a single input parameter), and each internal node a function on its child node(s). The root node is therefore recursively associated with a single function on the set of primitives (see Figure 3).

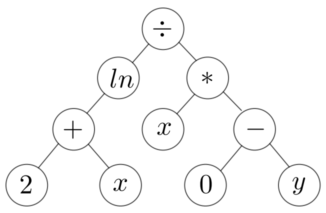

Figure 3: A chromosome in tree format representing the function $f(x, y) = \frac{\ln(2+x)}{x(0-y)}$.

Mutation is represented by pseudo-randomly replacing a node with a different function or primitive (or one of another set of mutation-like actions), as appropriate. Crossover between chromosomes is enacted by swapping sub-trees.

For the games in Sections 2 and 3 we examined a number of different sets of hyper-parameters, and the most reasonable for both convergence and computation time were heap sizes up to 10 on positions with anywhere from one to five heaps.

The package gpLearn is intended for fitting real-valued functions of several real variables to data points using standard elementary functions and binary operations over the reals. We modified the default set of functions to instead focus on discrete functions of several discrete variables. We wrote and included the following binary and unary operations, which operate bitwise on integer inputs: XOR, AND, OR, NOT. We also included MOD, LOG_2, and PLUS1, whose operations are self-explanatory. Finally, we introduced logical operators EQUAL, LESS, and GREATER to return 0 for False and 1 for True. Default functions included SUB for subtraction, ADD, TIMES, and DIVIDE. Our fitness function computed the total absolute difference between each genetic program and the computed Grundy values, so that a lower fitness value represents a better fit, although we experimented with measuring distance using the nim-sum.

Populations ranged from 1,000 to 10,000 individual programs, and we restricted most runs to 20 generations. *Elites*, relatively highly fit programs in each generation, were retained unmodified between generations. We also experimented with rates of mutation, settling on higher values to prevent getting stuck in local minima.

2. A Single-point Crossover and Mutation Game

There are two primary methods of crossover used in genetic algorithms, *one-point* and *two-point*. For the former, consider a pair of bit strings of length n, $B_1 = (a_1, \dots, a_n)$ and $B_2 = (b_1, \dots, b_n)$. An integer $k \in [1, n)$ is chosen pseudo-randomly, and the sub-strings $(a_1, \dots, a_k)$ and $(b_1, \dots, b_k)$ are swapped, leading to the new bit strings

$$B_1' = (b_1, \dots, b_k, a_{k+1}, \dots, a_n), B_2' = (a_1, \dots, a_k, b_{k+1}, \dots, b_n).$$

After crossover there is a possible mutation, depending on the chosen mutation rate, turning, say, B_1' into $B_1'' = (b_1, \dots, b_{i-1}, 1 - b_i, b_{i+1}, \dots, b_k, a_{k+1}, \dots, a_n)$.

Motivated by these processes we define a new impartial combinatorial game, GA1. In order to simplify both rules and analysis we define a position as a single bit string. A mutation move flips a single bit in the string, and while there is no real crossover in a single string we consider the flip of a sequence of bits to be representative of this operation.

Ruleset 2.1 (GA1)**.** A position in GA1 is a bit string of length n. There are two move options. Crossover consists of choosing an integer k, $1 \le k \le (n-1)$, wherein all bits from position 1 through k are flipped. A mutation move is simply the flip of any single bit in the string. A move is legal only if the total number of sub-strings of the form 01 and 10 increases.

This latter restriction, that 'disorder' increases, serves two purposes. Firstly, it ensures that the game ends in a finite number of moves. Secondly, it represents the tendency of chromosomes to combine in ever more complex ways over time. We define the condition of increasing sub-strings 01 and 10 formally as follows.

Definition 2.2. The *entropy* of a bit string game is the number of sub-strings of the form 01 and 10.

The game GA1 is equivalent to a heap game in the following way. If we consider a *run* in a bit string to be a maximal sub-string consisting of all 0s or all 1s, then any bit string can be converted into a list of integers representing run sizes. For example, the string 001000111 becomes $(2, 1, 3, 3)$. Although this representation loses information about which bits are associated with each integer, the symmetry of the ruleset makes this lost information unnecessary to the game analysis. We can simplify the ruleset further by Proposition 2.3.

Proposition 2.3. *If $H = (h_1, \dots, h_k)$ is a list of heaps representing a bit string in* GA1*, then the following hold.*

1. *Any heap equal to 1 can be removed.*
2. *The order of the heaps does not affect the Grundy value.*
3. *Each move is equivalent to one of the following.*
   - (a) *Split any heap $h_i > 3$ into two heaps of size at least 2 each.*
   - (b) *Remove 1 from any heap $h_i \geq 3$.*
   - (c) *Remove 1 from any heap $h_i \geq 5$ and split the remainder into two heaps of at least 2 each.*
   - (d) *Remove any heap of size 2 or 3.*

Proof. We will prove each part of Proposition 2.3 separately.

1. A single bit between two runs of the opposite value or at the end of a string is represented by a heap of size 1. No move that increases entropy has an effect on this heap, and thus its removal has no effect on game play nor the Grundy value of the position. Thus it can be removed.

2. Say that heaps h_i and h_j switch positions in H resulting in H'. Any mutation or crossover point k chosen within h_i in H is equivalent to the index k' in H', where $k' = k - (h_j + h_{j+1} + \cdots + h_{i-1})$ if h_j precedes h_i in H, and $k + (h_{i+1} + \cdots + h_{j-1} + h_j)$ otherwise. This results in identical game play. Therefore the order of the heaps in H does not affect the game value.

3. For the move equivalences, note that in order to make a legal crossover move in GA1 a player must choose the crossover point in the midst of a run. This effectively splits the run into two, and leaves the others alone (other than switching the bits in the affected sub-string). This is equivalent to splitting a heap into two and if one or both of the resulting heaps have size 1 then they can be removed from play. Similarly, a legal mutation move must also occur in the midst of a run, splitting a heap into either two or three with at least one heap of size 1. Again, these size 1 heaps can be removed.

□

As a direct result of Proposition 2.3 we need only consider single heap positions, since the Grundy value of a list of heaps is equal to the nim-sum of the Grundy values of the individual heaps.

The package `gpLearn` was employed as described in Section 1.1. In particular, game values were computed using positions of the form $(h_1, h_2, \ldots, h_n)$ with varying values for the number of heaps n and each heap h_i. These solutions were set as ground truth for the genetic programming implementation. The system generated random chromosomes, each associated with a function from $\mathbb{N}^n$ to $\mathbb{N}$, and utilized crossover and mutation to progressively reduce the error between the values of these functions and the ground truth values. While no exact formula was found, after 14 generations a local minimum on the total absolute error was reached. Modifying hyper-parameters and running for another 7 generations led to the formula

```
MOD(1+h,MOD(h+1,3) + 1) - MOD(h-1,4) + MOD(1+h,3) + 4
```

where h is the size of a single heap and `MOD(x,n)` represents $x \pmod{n}$. While not a particularly accurate formula, we do see the presence of both *modulo 3* and *modulo 4*. This leads us to examine the exact computed Grundy values closely for periodicity of order twelve and find a striking similarity with the values of the combinatorial game KAYLES.

Ruleset 2.4 (KAYLES [9])**.** In KAYLES a player may remove one or two stones from any heap, and if any stones remain these may be split into two heaps.

KAYLES has octal code 0.77 [8] and has been well-studied. In particular, it is known that the Grundy values for a single heap game of KAYLES of size n is periodic with period 12 after $n = 71$ [12].

Theorem 2.5. *The Grundy value of a single heap game of size n in* GA1 *is equal to the value of a heap of size $(n-1)$ in* KAYLES.

Proof. This is easy to compute for $n \leq 3$. If $n \geq 4$ then the options are $\{(j, n-j) : 2 \leq j \leq (n-j)\} \cup \{(k, n-k-1) : 2 \leq k \leq n-k-1\} \cup \{(n-1), (n-2)\}$. The options for an $(n-1)$-sized heap in KAYLES are $\{(j, n-j-2) : 1 \leq j \leq (n-j-2)\} \cup \{(k, n-k-3) : 1 \leq k \leq n-k-3\} \cup \{(n-2), (n-3)\}$. We can therefore consider a move in GA1 to be equivalent to the following process:

1. Remove a stone from a heap,
2. Make a KAYLES move in the resulting heap of size $(n-1)$,
3. Add a stone back to all resulting heaps.

Therefore the game GA1 reduces to a game of KAYLES, and thus the Grundy values are computable in the same manner as those for KAYLES. $\square$

While our genetic programming method did not determine a precise error-free formula for the Grundy values of GA1, it did inform our understanding of the game by pointing us to its periodic nature.

3. A Two-point Crossover Game

Next we consider a similar impartial game based on genetic crossover, this time using two positions instead of one. Consider a pair of bit strings

$$B_1 = (a_1, \ldots, a_n) \qquad \text{and} \qquad B_2 = (b_1, \ldots, b_n).$$

If $1 \leq x < y \leq n$ are integers then the *two-point crossover* using positions x and y results in the bit strings

$$B_1' = (a_1, \ldots, a_{x-1}, b_x, \ldots, b_{y-1}, a_y, \ldots, a_n)$$

and

$$B_2' = (b_1, \ldots, b_{x-1}, a_x, \ldots, a_{y-1}, b_y, \ldots, b_n).$$

That is, a sub-string with matching indices from each bit string is swapped. We wish to define an impartial game motivated by two-point crossover as a move mechanic. As we did with GA1, we play only in a single bit string. This also means that defining mutation-type moves is redundant since any such move would be equivalent to crossover with $x = y - 1$.

Ruleset 3.1 (GA2)**.** A position in GA2 is a bit string of length n. On their turn a player chooses two integers $x, y \in [1, n], x < y$, wherein all bits from position x through $(y - 1)$ are flipped. A move is legal only if the total number of sub-strings of the form 01 and 10 increases.

As with GA1 we can reduce GA2 to a game on heaps. Note that, again, a run of bits can be represented by an integer. A legal move requires that at least one of $\{x, y\}$ is chosen within a run. The possible options are

1. Both x and y are within the bounds of a single run, equivalent to splitting a single heap into three heaps,
2. x and y are each within the bounds of different runs, equivalent to splitting any two heaps into two each,
3. x and y are chosen so that exactly one single heap is split into two.

Just as with Proposition 2.3 we see that heaps of size 1 are negligible, as is the order of the heaps. However, since players can alter multiple heaps in a single move, we cannot compute the Grundy value by simply computing the nim-sum of the Grundy values of single heap games.

As in Section 2, we applied `gpLearn` with the modified function list to computationally determined Grundy values, without first examining these values. Once the number of non-zero heaps was included as a primitive value in games with more than two heaps (e.g. $(3, h_1, h_2, h_3)$ is a three-heap game, while $(4, h_1, h_2, h_3, h_4)$ represents a four-heap position), genetic programming proved much more successful, yielding the formulas below with 100% accuracy:

1. For a single-heap game with heap size h,
   `MOD(SUB(h,1),PLUS1(PLUS1(1)))`
   which is equivalent to $(h - 1) \pmod 3$.

2. With two heaps h_1, h_2
   `MOD(PLUS1(SUB(ADD(`h_1`, `h_2`), XOR(`h_1`, `h_1`))),`
   `PLUS1(PLUS1(EQUAL(`h_1`, `h_1`))))`
   which is equivalent to $(h_1 + h_2 + 1) \pmod 3$.

3. For a three-heap game with inputs $3, h_1, h_2, h_3$ we found
   `MOD(ADD(ADD(`h_3`, `h_1`), `h_2`), ADD(3, SUB(0, 0)))`
   which reduces to $(h_1 + h_2 + h_3) \pmod 3$.

While these results themselves do not provide a generalized formula, they do generalize easily to the following.

Theorem 3.2. *Let $H = (h_1, \ldots, h_n)$ be an n-heap position in* GA2*, and let t be the smallest non-negative integer such that $(n + t) \equiv 0 \pmod 3$. Then the Grundy value of H is $\left(t + \sum_{i=1}^{n} h_i\right) \pmod 3$.*

Proof. Note first that while we can eliminate heaps of size 1 in our analysis of GA2 just as we did in Proposition 2.3 for GA1, we are not compelled to do so. In fact, not removing them makes for a simpler analysis here.

In the case of a single stone it is clear that the Grundy value is 0 as no moves are possible. It is also easy to see that the claim holds when all heaps have size 1 except possibly a single heap of size 2, so we need only consider the remaining cases. We proceed now by minimum counter-example. Assuming that the claim is false, let m be the smallest integer such that not all games on m-many stones follow the statement of the theorem. Among all such games with m stones, let $H = (h_1, \ldots, h_j)$ be a position with the greatest number of heaps.

For a positive integer x, let $\{x_1, x_2\}$ and $\{x^1, x^2, x^3\}$ represent all sets of positive integers satisfying $x_1 + x_2 = x$ and $x^1 + x^2 + x^3 = x$. For any i, k with $1 \le i < k \le j$, the options of H are $H \setminus \{h_i\} \cup \{h_{i1}, h_{i2}\}$, $H \setminus \{h_i\} \cup \{h_i^1, h_i^2, h_i^3\}$, and $H \setminus \{h_i, h_k\} \cup \{h_{i1}, h_{i2}, h_{k1}, h_{k2}\}$; i.e., all positions in which any one heap, h_i, of sufficient size is removed and replaced with two or three heaps whose sum is h_i, and those in which any two heaps $h_i, h_k \geq 2$, are removed and each replaced with two heaps whose sums are h_i, h_k respectively.

All options contain m total stones since we have not removed any. Further, every option has more heaps than does H, and therefore, by the choice of minimal counter-example, adhere to the statement of the claim. Thus their Grundy values are all equal to $(m + t - 1) \pmod 3$ and $(m + t - 2) \pmod 3$. If there is a heap of size at least three then it can be split into two heaps via a crossover move or three heaps via mutation (changing only one bit). Similarly, if there are two heaps of size at least two then these can become four heaps through crossover or three heaps through mutation. Therefore H has options with one more and two more heaps, and hence both values $(m + t - 1) \pmod 3$ and $(m + t - 2) \pmod 3$, respectively, are present among the options. Thus, H must have Grundy value $(m+t) \pmod 3$, contradicting the choice of H as a minimal counter-example. □

Once again, as with GA1, our genetic programming implementation informed a conjecture about the Grundy values of GA2, which we were then able to prove. In this case the formula provided by the system was exact.

4. The Crossover-Mutation Game

In order to represent both crossover and mutation more accurately, we now consider a game played on a pair of bit strings.

Ruleset 4.1 (CROSSOVER-MUTATION (CM))**.** A position in CROSSOVER-MUTATION is a pair of bit strings of length n, $B_1 = (a_1, \ldots, a_n)$ and $B_2 = (b_1, \ldots, b_n)$. There are two move options. *Crossover* consists of choosing an integer k, $1 \leq k \leq (n-1)$, wherein all bits 1 through k from B_1 are swapped with the bits 1 through k of B_2. In particular, leading to the new bit strings

$$B_1' = (b_1, \ldots, b_k, a_{k+1}, \ldots, a_n), B_2' = (a_1, \ldots, a_k, b_{k+1}, \ldots, b_n).$$

Mutation involves choosing a single gene c_i from either of the bit strings and flipping it to $1 - c_i$. In both cases, the move is legal if the total number of sub-strings of the form 01 and 10 increases.

A position in CROSSOVER-MUTATION is composed of a pair of binary strings, in contrast to the single string for each of GA1 and GA2. This results in a higher dimensional domain space for game positions, even when binary strings are converted to heaps as above. Our genetic programming approach proved unable to account for so many inputs and, hence, we used standard CGT methods.

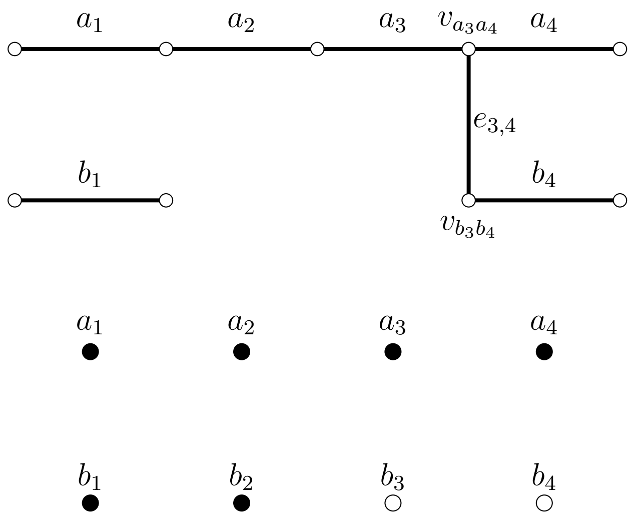

Figure 4: Example of an ARC KAYLES position which is equivalent to a position in CROSSOVER-MUTATION.

Firstly, we note a reduction to a known combinatorial game ruleset. All positions of CROSSOVER-MUTATION are equivalent to certain positions from another game called ARC KAYLES. We first present the ruleset, then prove the equivalence.

Ruleset 4.2 (ARC KAYLES [17])**.** Let G be a graph. On a player's turn, they remove an edge of G along with all edges incident to it.

Theorem 4.3. *Let G be a* CM *position. G is equivalent to an* ARC KAYLES *position.*

Proof. Let G_{CM} be a CM position of length n as $B_1 = (a_1, \dots, a_n)$ and $B_2 = (b_1, \dots, b_n)$. We will first construct the ARC KAYLES position, G_{AK}. Then we will prove its equivalence by showing that there is a bijection between the options of the games.

First consider B_1 of G_{CM}. For each mutation, its representation in G_{AK} is an edge. Edges are incident in G_{AK} if the corresponding bits in G_{CM} were adjacent in B_1. Similarly for B_2. We label the edges of G_{AK} by the corresponding bit labels in B_1 or B_2 respectively. For the crossover moves in G_{CM}, if there exists a crossover move at a_i, a_{i+1} and b_i, b_{i+1} then in G_{AK} there are vertices, call them $v_{a_i,a_{i+1}}$ and $v_{b_i,b_{i+1}}$, between respectively labelled edges. Denote the edge connecting $v_{a_i,a_{i+1}}$ and $v_{b_i,b_{i+1}}$ by $e_{i,i+1}$. See Figure 4 for an example of the equivalence in which we have represented the bits as colored and uncolored vertices.

To show that G_{AK} is equivalent to G_{CM} via this construction, we need to show that there exists a bijection between the options. In particular, that $G_{CM} - G_{AK} =$

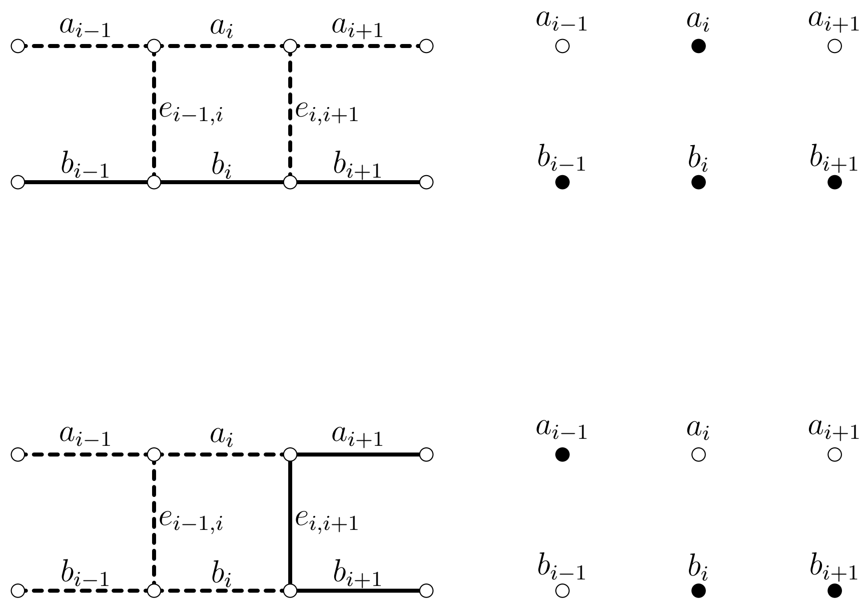

Figure 5: Mutation: The result of a mutation move at a_i in CROSSOVER-MUTATION (top right) and the corresponding resulting position after a move in ARC KAYLES (top left). Crossover: The result of a crossover move at $i-1$ in CROSSOVER-MUTATION (bottom right) and the corresponding resulting position after a move in ARC KAYLES (bottom left). Dashed lines represent the edges which were eliminated by moving in ARC KAYLES.

0. Since the rulesets are impartial, we consider $G_{CM}+G_{AK}$. Suppose the first player moves in G_{CM} with a mutation at a_i. By the existence of this mutation, it means that both a_{i-1} and a_{i+1} were the same as a_i (if they exist), otherwise the entropy would not have increased. After the turn, neither can be mutated thereafter because again, it would not increase the entropy. Also, this move disallows future crossover at a_i because it will not increase the entropy. Player 2 responds by removing the edge $a_i \in G_{AK}$. This has the effect of removing all incident edges, in particular, a_{i-1}, a_{i+1}, $e_{i-1,i}$, and $e_{i,i+1}$, if they exist (see Figure 5). If instead Player 1 chose a crossover move in G_{CM} at position $i-1$, this eliminates the possibility of future mutations at positions a_i, a_{i-1}, b_i, and b_{i-1}. The corresponding move for Player 2 is to respond in G_{AK} by removing the edge $e_{i-1,i}$, which also removes all edges a_i, a_{i-1}, b_i, and b_{i-1} (see Figure 5).

If instead Player 1 moved in G_{AK}, we simply reverse the roles in the above argument and Player 2 will always have a response. Thus Player 2 will win this game under normal play. Hence G_{CM} and G_{AK} are equivalent. □

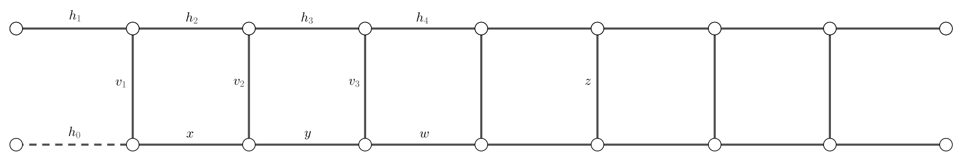

Figure 6: An ARC KAYLES position; equivalent to a position in CROSSOVER-MUTATION when h_0 is present. Here the dashed line represents an edge that may or may not be present.

If the CM position is of a certain form, in particular every entry a_i of B_1 is the same, and every entry of B_2 is $1 - a_i$, the proven equivalence to a subset of ARC KAYLES positions allows us to immediately deduce the game values.

Theorem 4.4 ([4])**.** *Let G be a position in* ARC KAYLES *in the form of a $2 \times n$ grid graph. Then G has value 0 if n even and value $*$ if n odd. Furthermore, this game value does not change under the addition of up to two* tufts *(i.e., induced stars whose center is a vertex of the grid graph).*

We can extend this result to find values for other CM positions.

Theorem 4.5. *Let $G(k)$ be a position in* ARC KAYLES *in the form of a $2 \times k$ grid graph with pendant edges adjacent to 3 or 4 of the four corners (see Figure 6). Then $G(2k+1)$ has game value $*2$ if $k \in \{0, 1\}$ and $*$ if $k \geq 2$, and $G(2k)$ has value 0 for all $k \geq 1$ when h_0 is present.*

Proof. Note that if $k \leq 1$ then the possible values of $G(2k+1)$ are easily demonstrated by exhaustion. The value of $G(2k)$ is just as easily found to be in $\mathcal{P}$ by considering an involution strategy, whereby the second player responds to a play on edge e with a play on the edge equivalent to e under $180°$ rotational symmetry. We now proceed by induction on k to find the remaining values of $G(2k+1)$ whether or not edge h_0 is present.

Let e be an edge in $G(2k+1)$, and consider $H(e)$ to be the option yielded by play on e (see Figure 7). We demonstrate that no option of $G(2k+1)$ has value $*$.

$H(h_1)$: Play on edge x results in a graph of the form $2 \times (2k-1)$ with three pendant edges. If k is sufficiently large this graph has value $*$ by inductive assumption, and hence $H(h_1)$ does not have value $*$. Otherwise, the value can be checked exhaustively for the base case of $G(5)$, when $k = 2$, to have value $*$ with or without the

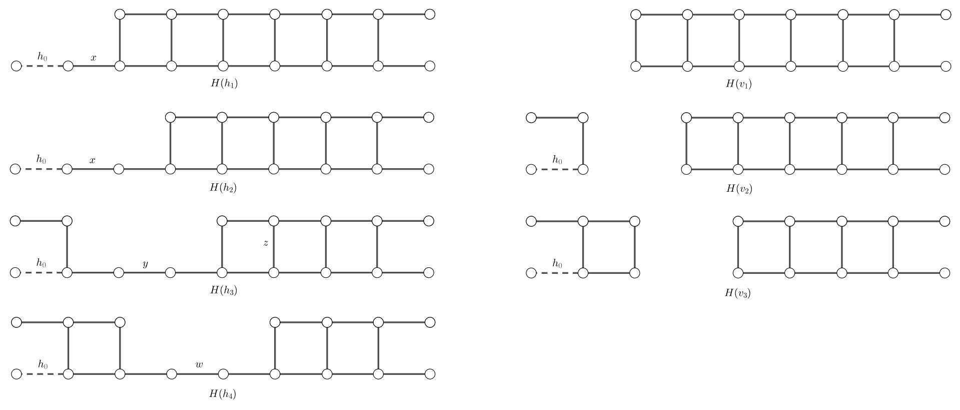

Figure 7: The options of $G(2k+1)$ from Figure 6.

presence of h_0. Hence, $H(h_1)$ does not have value $*$.

$H(h_2)$: Play on the edge x results in a position with value $*$ by Theorem 4.4. Therefore $H(h_2)$ does not have value $*$.

$H(h_3)$: If h_0 is not present then play on edge y yields a path with value $*$ disconnected from a $2 \times (2k-2)$ grid graph with two pendant edges which, by Theorem 4.4, has value 0. If h_0 is present then play on edge z yields the sum of a small graph with value $*$ and a $2 \times (2k-4)$ grid graph with two pendant edges. In both cases, the resulting sums are $*$. Therefore, $H(h_3)$ does not have value $*$.

$H(h_4)$: Here h_4 can be any horizontal edge to the right of h_3. Play on edge w results in a game with a sum of two positions with opposite parity. Hence has value $* + 0 = *$ by Theorem 4.4, so $H(h_4)$ does not have value $*$.

$H(v_1)$: This graph has value 0 by Theorem 4.4.

$H(v_2)$: If h_0 is present then we have the sum of a path with value $*2$ and a game with value $*$ by Theorem 4.4. If h_0 is not present then the path has value $*$. So $H(v_2)$ has value $*3$ or 0.

$H(v_3)$: We invoke Theorem 4.4 yet again, as the resulting graph is a pair of grid graphs with one or two pendant edges each, both with value $*$ or both with value 0. Therefore $H(v_3)$ has value 0.

Since no option of $G(2k+1)$ has value $*$ and $G(2k+1) \in \mathcal{N}$, we see that it has value $*$ for $k \geq 2$. $\square$

Theorem 4.5 leads directly to the following corollary about a family of CROSSOVER-MUTATION positions.

Corollary 4.6. *The* CM *game composed of a length-n string of all* 1*s and a length-n string of all* 0*s has value* 0 *if n is odd,* $*2$ *if $n \in \{2,4\}$, and* $*$ *otherwise.*

Proof. This position is equivalent to the ARC KAYLES position $G(n-1)$ with h_0 present, as indicated in Theorem 4.5. $\square$

It turns out that CM is also closely related to another well-studied game.

Ruleset 4.7 (CRAM [4])**.** In the impartial game CRAM players take turns filling a pair of empty orthogonally adjacent spaces in a grid. See Figure 8.

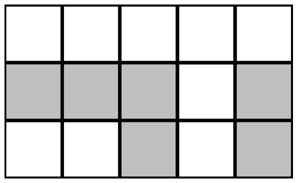

Figure 8: Example of a 3×5 CRAM position that is equivalent to the ARC KAYLES and CM positions in Figure 4.

The reader may recognize CRAM as the impartial version of DOMINEERING. All CM positions are also associated with $2 \times n$ CRAM positions, except for a few with extra pendant edges which, if realized in CRAM, require a board of width at least three. This association is produced by first considering the associated ARC KAYLES position. See Figure 8 for a CRAM position which is equivalent to the ARC KAYLES position pictured in Figure 4. Note that this position cannot be realized by a $2 \times n$ CRAM position, for any n.

To a CM position (B_1, B_2) with $B_1 = (a_1, \ldots, a_n), B_2 = (b_1, \ldots, b_n)$, we associate an ARC KAYLES position as above, resulting in a subgraph of a $2 \times (n-1)$ grid graph with up to four pendant edges at the corners. A $2 \times n$ grid graph in ARC KAYLES is equivalent to a $2 \times n$ grid in CRAM, as the removal of a vertical (horizontal) edge and its neighbors relates to adding a vertical (horizontal) game piece to the CRAM board. A single vertex missing from this position is equivalent to a blocked square in the associated grid, and the pendant edges associate to extra spaces that each share an edge with one of the four corner spaces, without sharing edges with any other spaces.

Most remaining CM positions are equivalent to $2 \times n$ positions in CRAM which, while remaining unsolved, have been addressed in the literature [4]. It is worth noting that all CM positions in which no crossover move is possible are simply represented by a disjunctive sum of paths in ARC KAYLES, whose values are known [15].

5. Conclusion and Further Research

We have seen the possible application of genetic programming to the determination of Grundy values of impartial combinatorial games. In addition, we have seen it both provide an exact function and simply inform our own mathematical analysis. Note that the game for which it proved most useful, GA2, could likely have been solved without the use of genetic programming and instead through a simple examination of the computed Grundy values. But we have also seen that it *was* solved through the use of genetic programming, and therefore this method could prove useful in the future. At the very least, it could be utilized to reduce the time and effort taken to conjecture formulas for Grundy values.

We are curious whether or not genetic programming can be used for problems within CGT that a mathematician simply examining a list of values is unlikely to solve. To answer this we suggest more efforts into this practice. It will be very useful, for example, to compile a database of impartial combinatorial games with known and as yet unknown solutions. This could help inform the choice of default functions to include in future genetic programming attempts.

There are modifications that we suggest be made to future GP for CGT projects. Firstly, it would be beneficial to develop a more robust fitness function. As there is no obvious metric over the set of nimbers outside of the nim-sum, an analytical approach to metrics over impartial games would be helpful. Secondly, the method for fitness employed in [16] does not use pre-computed data points at all. Instead the author determines the fitness of a program by comparing the computed outcome classes of a set of positions with those of its options, and relating the fitness to the number of deviations from the basic tenets of impartial games that are found among these computations. Something similar could be used for Grundy value programming, involving the *mex* (minimum excludant) function. However, the distance between actual value and computed value remains a possible stumbling block.

References

[1] M. H. Albert, R. J. Nowakowski, and D. Wolfe, *Lessons in Play: An Introduction to Combinatorial Game Theory*, A K Peters, Ltd., Wellesley, MA, 2007.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Volume 1, second edition, A K Peters, Ltd., Natick, MA, 2001.

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Volume 2, second edition, A K Peters, Ltd., Natick, MA, 2003.

[4] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Volume 3, second edition, A K Peters, Ltd., Natick, MA, 2003.

[5] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Volume 4, second edition, A K Peters, Ltd., Wellesley, MA, 2004.

[6] A. Bradford, J. K. Day, L. Hutchinson, B. Kaperick, C. E. Larson, M. Mills, D. Muncy, and N. Van Cleemput, Automated Conjecturing II: Chomp and Reasoned Game Play, *J. Artificial Intelligence Res.* **68** (2020), 447-461.

[7] R. M. Brady, Optimization strategies gleaned from biological evolution, *Nature* **317** (1985), 804–806.

[8] J. H. Conway, *On Numbers and Games*, second edition, A K Peters, Ltd., Natick, MA, 2001.

[9] H. E. Dudeney, *The Canterbury Puzzles (and Other Curious Problems)*, EP Dutton, New York, 1908.

[10] P. Galinier and J.-K. Hao, Hybrid evolutionary algorithms for graph coloring, *J. Comb. Optim.* **3** (1999), 379–397.

[11] P. M. Grundy, Mathematics and games, *Eureka* **2** (1939), 6–8.

[12] R. K. Guy and C. A. B. Smith, The G-values of various games, *Math. Proc. Cambridge Philos. Soc.* **52** (3), (1956), 514–526.

[13] A. Hauptman and M. Sipper, Analyzing the intelligence of a genetically programmed chess player, In *Late Breaking Papers at the Genetic and Evolutionary Computation Conference 2005.* Washington DC, June 2005.

[14] G. Hornby, A. Globus, D. Linden, and J. Lohn, Automated antenna design with evolutionary algorithms, *Space 2006*, 2006.

[15] M. Huggan and B. Stevens, Polynomial time graph families for Arc Kayles, *Integers* **16** (2016), #A86.

[16] M. Oltean, Evolving winning strategies for Nim-like games, *IFIP Student Forum* (2004), 353–364.

[17] T. J. Schaefer, On the complexity of some two-person perfect-information games, *J. Comput. System Sci.* **16** (1978), 185–225.

[18] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel, Designing an evolutionary strategizing machine for game playing and beyond, *IEEE Transactions on Systems, Man, and Cybernetics, Part C* **37** (4), (2007), 583–593.

[19] R. Sprague, Über mathematische kampfspiele, *Tohoku Mathematical Journal, First Series* **41** (1935), 438–444.

[20] T. Stephens, *Gplearn Model, Genetic Programming*, Copyright, 2015.