Finding a Generalization of Nim-Values for 3-Player Impartial Combinatorial Games with Fixed Winning Preferences

Sammy Luo, Kavi Jain, Tejas Sundaresan*

February 26, 2013

Abstract

In this paper we explore the properties of 3-player impartial combinatorial games with fixed winning preferences. We begin our observations by finding patterns and proving results for the n-player game of Nim. We find that the winning strategy in the n player game is based on the notion of adding pile numbers in binary and interpreting them as base n integers. We then attempt to extend these results and generalizations to combinations of subtraction games and Nim. As far as subtraction games, we discuss a few preliminary conjectures about the generation of nim-values in the games, find a few results on periodicity of different games, and eventually find a way to generate the nim-values using nim-pairs. To gather and record results for the combinations of games, we use java programs and Mathematica software. After data analysis, we report our findings on the various properties of these games and describe future work and limitations of the problem, including accounting for anomalies and finding a probabilistic way to determine the frequency of anomalous results.

1 Introduction

1.1 Combinatorial Games

The game of *Nim* is an impartial game where given one or more piles, players alternate taking away at least one counter from exactly one of the piles. The winner of Nim is the person who takes the last counter or stack of counters[1]. The Sprague-Grundy theorem states that any (two-player) impartial game is equivalent to some game of Nim. Since 2-player Nim is completely solved, this implies that a definite solution (winning strategy) exists for any such game.

Another type of impartial game is the subtraction game, which is played in a similar fashion. However, it is distinguished from Nim because in the game, players may only remove a certain number of stones, $k \in S$, where S denotes the *subtraction set* for the game.

*We would like to thank our mentor, Dr. Daniel J. Teague, for helping us throughout this project.

For example, in the subtraction game $\{1, 3, 7\}$, players may only remove 1, 3, or 7 stones from a pile on each turn.

The *nim value* (a.k.a. Grundy number) of a game position is defined recursively as the smallest nonnegative integer that is not the nim-value of a position it can reach in one step, with the nim-value of the end position(s) being 0. A position is losing if and only if it has nim-value 0. Accordingly, nim-values are a generalization of the idea of winning (W, or next-player-wins, N) positions and losing (L, or previous-player-wins, P) positions.

Nim-values simplify the process of analyzing combinations of games due to the fact that the nim-value of the sum of two games is the *nim-sum* of the two games' nim-values, i.e. the result of summing corresponding digits in the two games' nim-values mod 2. This allows us to find winning strategies for sums of many individual games as long as we can compute the nim-values for the individual games.

The nim-sum is denoted by the $\oplus$ operator. It satisfies all the necessary properties of addition over an abelian group (commutativity, identity, inverse, etc.).

The simplest way to discover the applicability of the nim-values and the nim-sum as defined to game sums is to derive them from the game of Nim itself, which (since a pile of n stones simply represents a nim-value of n) allows us to easily discover what operation to perform when summing games.

A natural question after finding that all impartial 2-player games are solvable is: what about games with more than 2 players? With many players, the situation becomes much more complex; rarely ever can any player guarantee a win over the others regardless of the moves of the other players [2]. As such, for most games there is no satisfying "solution" in the sense of allowing a player to guarantee a win when playing the game with strangers in real life. However, in using certain assumptions, we were able to devise a strategy for 3-player, and more generally, n-player Nim.

1.2 The Fixed-Preference 3-Player Generalization

Let the three players be players 1, 2, and 3; their moves cycle in that order, starting with player 1. These players play a 3-player analog of some 2-player impartial combinatorial game (for example, Nim or some subtraction game). The first player who cannot make a move loses. Two assumptions are key to our generalization:

(a) Each player wants above all to avoid losing, but given the choice would prefer the next person to lose, rather than the previous person, and

(b) All players (besides possibly the loser) play optimally. (The loser's move choices do not affect the outcome of the game as long as other players move optimally.)

Condition (a) seems natural in the real world (at least, more natural than the reverse direction) because it's reasonable for one to want to be as close before the losing player as possible without losing (to have the most recent contribution to defeating that player).

Condition (b), however, does not seem as natural. It implies that our theory of winning positions can only work in the real world if the winner W can convince all other players (except the losing player) that the best the k-th player before W can do is get $k+1$-th place and that doing that involves following W's instructions. However, in the mathematical

context, this is quite a natural generalization (just as natural, perhaps, as the assumption that all players besides possibly the winner can do whatever they want, which is basically symmetric to (b)) of impartial games to more than 2 players, and allows us to find some meaningful results.

In this generalization, winning positions are denoted A positions, losing positions are denoted C positions, and "middle" positions are denoted B positions. These positions are then recursively defined as follows: the end position(s) is a (are) C position(s); anything leading to a C position is an A position; anything that's not an A position but leads to one is a B position; anything else is a C position.

Letting $C = 0$, $A = 1$, $B = 2$ gives that the A/B/C value of a position is $(\min(S) - 1) \pmod 3$, where S is the set of A/B/C values of positions it can reach. We then try to find a generalization of the nim-value (a 3-nim value) that corresponds to this.

Denote by n_G the (hypothetical) 3-nim value of a game position G. Ideally, we will find such an assignment of n_G to game positions and a binary operation $\boxplus$ on 3-nim values such that:

(1) The set of possible n_G and the operation $\boxplus$ form an abelian group, with $n_0 = 0$ as the additive identity.

(2) For some sets X, Y, Z we have that G is an A-position iff $n_G \in X$, a B-position iff $n_G \in Y$, and a C position iff $n_G \in Z$.

2 Results on m-Nim

To find some patterns that may lead to discoveries, we created programs in Java and used Microsoft Excel to generate and tabulate 3-player data for some games for small starting positions. After proving that a position $P = (x_1, x_2, \cdots, x_n)$ in 3-Nim is a C-position if and only if

$$(x_1)_T \boxplus (x_2)_T \boxplus \cdots \boxplus (x_n)_T = 0$$

where x_T, the *ternation*, is the reading of the base 2 representation of x as a base 3 integer, and $\boxplus$ is the 3-sum, the digitwise sum $\pmod 3$ of the arguments in base 3. We now introduce a way to differentiate between A and B positions.

Proposition 2.1. *Consider a position $P = (x_1, x_2, \cdots, x_n)$ in 3-Nim, and let $s_3(P) = (x_1)_T \boxplus (x_2)_T \boxplus \cdots \boxplus (x_n)_T = \overline{c_{m-1}c_{m-2}\cdots c_1c_0}_3$ in base 3 with c_{m-1} nonzero. Then P is an A-position if and only if the following conditions hold:*

(i) $c_{m-1} = 1$, and

(ii) For some i, we have $x_i = \overline{a_{m-1}a_{m-2}\cdots a_1a_0}_2$ such that for all k, $a_k = 0$ if $c_k = 2$, and $a_k = 1$ if $c_k = 1$. (When $c_k = 0$, a_k can be either 1 or 0.)

Proof. By definition, P is an A-position if and only if replacing exactly one of the x_i with $x_i' < x_i$ yields a C-position P', so $s_3(P') = 0$. Let $s_3(P') = \overline{c'_{m-1}c'_{m-2}\cdots c'_1c'_0}_3$. Notice that $c_k \equiv \sum_{i=1}^{n} a_k(i) \pmod 3$, where $a_k(i)$ is the $(k+1)$-th digit from the right in x_i. $c_k' - a_k' \equiv c_k - a_k \pmod 3$ for all k. When the conditions in the statement hold, we can

clearly just change x_i as follows: when $c_k = 1$, make $a'_k = 0$, and when $c_k = 2$, make $a'_k = 1$. The leftmost digit changed is $a_{m-1} = 1$, which is made into $a'_{m-1} = 0$, so clearly x_i is decreased and we get $s_3(P') = 0$ as wanted. Similarly, if there is some x_i that can be changed to make $s_3(P') = 0$, condition (ii) must hold; otherwise, since all digits of x_i, x'_i are 0 or 1, the needed changes in the c_k cannot be executed. In addition, if condition (i) did not hold, we would have $c_{m-1} = 2$, so we need to increase a_{m-1}. Since x_i must be decreased, we would need to change a_k for some $k > m-1$, which would then cause c_k to become nonzero, contradiction. So the conditions are necessary and sufficient for P to be an A-position. $\square$

The above results essentially solve 3-Nim. We now wish to generalize these results for m-Nim. Below we prove the general theorem regarding losing positions for m-Nim.

Theorem 2.2. *In m-Nim, m players each try to have the most recent play possible at the end of the game (without being the person unable to make a move). Assuming everyone plays perfectly, a player being given the position $P = (x_1, x_2, \cdots, x_n)$ in m-Nim will lose (be the first person unable to play) if and only if*

$$(x_1)_M \boxplus (x_2)_M \boxplus \cdots \boxplus (x_n)_M = 0$$

Where x_M, the m-ation, is the reading of the base 2 representation of x as a base M integer, and $\boxplus$ in this case is the m-sum, the digitwise sum $\pmod m$ of the arguments in base m.

Proof. We use strong induction on $s := x_1 + x_2 + \cdots + x_n$. The theorem statement is trivial for $s = 0$ by the definition of a losing position. Assume for some $k > 0$ that the statement holds for all $s < k$. Consider a position $P = (x_1, x_2, \cdots, x_n)$ with $s = k$. Clearly any move will decrease s.

Let $s_M(P) = (x_1)_M \boxplus (x_2)_M \boxplus \cdots \boxplus (x_n)_M$. We will prove and use the following lemma:

Lemma 2.3. *Any position P with $s_M(P) \neq 0$ can be moved to one with $s_M(P) = 0$ in at most $m-1$ moves, while a position P with $s_M(P) = 0$ cannot be moved to another such position in at most $m-1$ moves.*

Proof. For the first part of the lemma: Let $s_M(P) = \overline{c_{d-1}c_{d-2}\cdots c_1c_0}_M$ in base M with c_{d-1} nonzero. If we can show that the lemma is true when the sizes of all piles have $\leq d$ digits in base 2, then in any other case we can ignore all but the last d digits of each pile and perform the same operation on these digits to get the same result.

Let the largest m piles in P have sizes $a_1 > a_2 > \cdots > a_m$. We are assuming that a_1 has at most d digits in base 2. Let $c_{d-1} = j$. We use strong induction on d to show that it is possible to perform the operation specified in the lemma with all of $a_1, a_2, \cdots, a_j$ changed. For $d = 0$, this is trivial because all the nonempty piles are 1s, so we remove the c_0 biggest piles and are done. Now assume it's true for all $d < d_0$ for some $d_0 > 0$, and consider $d = d_0$.

Let $c_{d0-1} = j_0$. Then replace $a_1, a_2, \cdots, a_{j0}$ with $a^* = 2^{d_0-1} - 1$. This results in a position P' with $d = d' \leq d_0 - 1$ (since c_{d_0-1} is now 0), so by induction the $c'_{d'-1}$ piles whose last d' digits form the largest base 2 integers possible can be used to do the operation demanded by the lemma. Since all digits of a^* in base 2 are 1, clearly $a_1, a_2, \cdots, a_j$, which have all been replaced by a^*, can be used from now on for any number of digits $d < d_0$ until more than j_0 digits are needed for some future digit, in which case we continue using them along with as many of the next largest integers as necessary.

So the statement holds for $d = d_0$ as well, and by strong induction holds for all d.

Now we prove the second part of the lemma. Assume the opposite, so there are piles $a_1, \cdots, a_{m-1}$ in P such that there exist nonnegative integers $a'_1 < a_1$, $a'_2 \leq a_2$, $\cdots$, $a'_{m-1} \leq a_{m-1}$ with $(a_1)_T \boxplus (a_2)_T \boxplus \cdots \boxplus (a_{m-1})_T = s_M(P) \boxplus (a'_1)_T \boxplus (a'_2)_T \boxplus \cdots \boxplus (a'_{m-1})_T = (a'_1)_T \boxplus (a'_2)_T \boxplus \cdots \boxplus (a'_{m-1})_T$. Since the digits of the $(a_i)_T$ and $(a'_i)_T$ in base m are 0s and 1s, summing $m-1$ of them cannot produce "'carry-overs"', so $(a_1)_T \boxplus (a_2)_T \boxplus \cdots \boxplus (a_{m-1})_T = (a_1)_T + (a_2)_T + \cdots + (a_{m-1})_T$ and $(a'_1)_T \boxplus (a'_2)_T \boxplus \cdots \boxplus (a'_{m-1})_T = (a'_1)_T + (a'_2)_T + \cdots + (a'_{m-1})_T$. But then we get $(a_1)_T + (a_2)_T + \cdots + (a_{m-1})_T = (a'_1)_T + (a'_2)_T + \cdots + (a'_{m-1})_T$, which is impossible if $a'_1 < a_1$ and $aA'_i \leq a_i$ for $i > 1$ (since it is simple to see, e.g. by expanding out the integers' base representations into polynomials of the base, that $a > a'$ implies $a_T > a'_T$, etc.). This completes the proof of the lemma. □

Now we continue with the main proof. If $s_M(P)$ is nonzero, then there exist piles $a_1, \cdots, a_j$ with $j \leq m-1$ that can be altered to produce a position P' with $s_M(P') = 0$. Pick such a list of piles with j minimized. Then the player (WLOG player 1) who is faced with position P can alter a_1 as required. If player i for $i = 2, 3, \cdots, j$ alters a_i as needed, they can get player $j+1$ to lose; since all of them will prefer player $j+1$ losing over any other player who they can possibly get to lose (including player 1) losing, perfect play implies that they will not allow player 1 to lose. So P is not a losing position.

If instead $s_M(P) = 0$, notice that altering any pile will result in a position P' with $s_M(P') \neq 0$ (by, e.g., the lemma). P' is then not a losing position. However, by the lemma applied to P, P' cannot be moved to a position P'' with $s_M(P'') = 0$ in $\leq m-2$ moves, and since by the inductive hypothesis such positions are the only losing positions, P' cannot be less than $m-1$ steps before a losing position, so it must be $m-1$ steps before a losing position, which makes P a losing position as claimed.

So the theorem holds for $s = k$ as well, and therefore by induction for all s. □

3 Observations on Subtraction Games

3.1 Conjectures Regarding Periodicity

Conjecture 3.1. *Given subtraction set $S = \{a_1, .., a_{n-1}\}$ with period p for the A/B/C position sequence, a subtraction game with subtraction set $S' = \{a_1, .., a_n\}$ where $a_n \equiv a_i \pmod{p}$ and $1 \leq i \leq n-1$ has the same position sequence and same period p. If $a_n \not\equiv a_i \pmod{p}$, the period p' of the new subtraction set is some member of the set $S' + S' + S'$ that is not contained in $S' + S'$.*

The first part of the conjecture is actually proven in the results section below, but the second part is a little harder to prove. At first, we conjectured that the period is simply the smallest member of the set $S' + S' + S'$ that isn't a member of $S' + S'$ but that doesn't always hold. However, we have yet to find a case where the period isn't in this set of values.

3.2 General Conjectures about Nim-Values

In the Results section below, we prove some interesting ideas about generalizing nim-values for subtraction games. However, in this section, we introduce some of our initial

conjectures and their relative success.

Conjecture 3.2. *In any subtraction game, we can recursively define nim values by a covering algorithm. Essentially, we take any position and look at the nim-values of all positions that this position can lead to in 1 turn. We now use a covering nim-value depending on the circumstances in order to ensure that the nim-value of a position along with the nim-values of positions it can reach form a set of consecutive integers in binary. The special thing about this covering is that we use 2s (in the base 3 nim-values) as 1s or 0s in order to cover the integers in the consecutive integers in binary. However, there are several conventions/conditions that must be followed:*

(i) If a covering is used, it must be the smallest covering possible.

(ii) A nim-value cannot be equal to a nim-value of a position that can be reached in at most 2 turns.

*(iii) Define a position to be **healthy** if it can reach 0, and **unhealthy** if it cannot reach 0 and is not 0. If a position leads to only unhealthy positions it is a 0.*

(iv) If the number being covered has the same number of digits as the nim-value of any other position that can be reached, its cover must have a larger number of digits (?)

(v) When possible, covers that have a digit of 2 should not use the 2 as a 0 in defining nim-values of higher positions, unless the whole number is being a 0 (?)

This conjecture seems to hold for all subtraction games with subtraction sets that are of the form $\{1, .., n\}$ or have cardinality at most 2. However, these conditions are a bit contrived and vaguely defined, so we attempt to find a more motivated solution using the **mex (minimal excluded value)** operator.

Conjecture 3.3. *In any 3-player subtraction game, we can also define nim-values using the mex of all the nim-values of the positions it can reach in 1 or 2 moves. However, to explain anomalies, we need a few conditions.*

(i) If the positions you can reach in 1 turn include nothing with digit 2 and do not include value 0, you do a base 2 mex of all the values that have a 2 or 0 in them to generate the next nim-value. Essentially, we treat the 2's as 1's and the output is represented by 0's and 2's (the 2's represent 1's).

(ii) If the positions you can reach in 1 turn include nothing with digit 2 but include value 0, you do a base 2 mex of all the values that have a 1 or 0 in them to generate the next nim-value.

(iii) Otherwise, just take a normal mex in base 3.

Even with the conditions, there are still anomalies in the mex-generated sequence. In many cases it seems that the mex conjecture (with the conditions) holds as soon as the nim-value sequence starts to become periodic; the discrepancies between the actual nim-values and the nim-values generated using the mex algorithm seem to only occur before the sequence becomes periodic. However, the success we had with this conjecture led us to further explore the mex operator, leading to some results of the next section.

4 Results on Subtraction Games

Definition 1. For any game G, let G_k be the game position of G with k stones in a pile. Thus, N_k is the nim-pile with k stones, while S_k (where the subtraction set for S has already been defined) is a subtraction pile with k stones.

Definition 2. Adding games is denoted by $+$, and losing positions can be set equal to 0 (e.g. $N_k + N_k + N_k = 0$)

Definition 3. $N_v(G_k)$ is the (3-)nim value for G_k.

4.1 Results Involving Periodicity

Theorem 4.1. *In any 3-player subtraction game with subtraction set $\{1, .., n\}$, the period length of the A/B/C positions will be $2n + 1$.*

Proof. Let us prove by construction. The first position is a C position, then we have a string of n A positions because these can all reach the losing position, then we no longer can reach the first position so we have a string of n middle positions (all of which can reach the A positions but not the C position). Then, we can only reach the B positions, so the next position is a C position. Then, because the next n positions can reach the C position they are all A positions, and the cycle repeats mod $2n + 1$. □

We would also like to prove a little bit more regarding periodicity of subtraction games. We note the following intuitive, but helpful, theorem.

Theorem 4.2. *Given subtraction set $\{a_1, .., a_{n-1}\}$ with period p for the A/B/C position sequence, a subtraction game with subtraction set $\{a_1, .., a_n\}$ where $a_n \equiv a_i \pmod p$ and $1 \leq i \leq n - 1$ has the same position sequence and same period p.*

Proof. We proceed by contradiction. Assume that adding a_n changes the nim sequence. We pick the smallest pile size, M, that gets changed. This means $M \geq a_n$ because a_n must be removable from it. However, we note that, by the given periodicity of the original subtraction set, if removing some $a_i \equiv a_n \pmod p$ from M gives a losing position, removing a_n from M also gives a losing position. So we have that $M - a_n$ iff $M - a_i$ is a C position. Similarly, $M - a_n$ iff $M - a_i$ is a A position, and the same holds for B positions. Because the ability to reach previously unreachable ABC values from M has not changed, M has the same ABC values after a_n is added. This contradicts our assumption that a_n affects M and we are done. □

4.2 Generalizing the Nim Value

For a 2-player game position G the nim-value is defined recursively as the mex of the nim-values of all positions it can reach, with the nim-value of terminal positions being 0; it also happens to be the size of the nim-pile that G is equivalent to in determining whether its sum with some other positions is a losing position.

To generalize this notion to 3-player games, we would like to find a function $N(G)$ that outputs the 3-sum of the nim position that G is equivalent to; this is determined by seeing

which nim-piles G 3-sums with to give 0, and 3-subtracting those piles from 0 (or, equivalently, adding each twice).

Unfortunately, as we have seen, the results are not very satisfying. For one, summing a 3-player game G to two different games A and B which are known to have the same nim-value (for example, if they are both nim games) does not always yield the same nim-value, and indeed can lead to different results for A/B/C position. In particular, $G+G+G$ is not always zero. In addition, some game positions G appear to have *multiple nim-values*! This is clearly not what a direct generalization of 2-Nim values would look like.

Upon further analysis, we see that the problem is with our way of looking at 2-Nim values in the first place. The 2-Nim value of G is the size of the nim-pile that G is equivalent to, but it is also the size of the nim-pile that sums with G to give 0. If we look at it this way instead, the generalization to 3 players would be to have *nim-pairs*, pairs of nim-piles that sum with G to give 0. This allows us to determine some definite formulas, which we can then (later) use to find the equivalent nim-value.

First we extend the definition of a mex to (unordered) pairs:

Definition 4. The *mex*es $\text{mex}(S_1, S_2)$ of two sets S_1, S_2 of pairs of nonnegative integers are the pairs $O = \{a, b\}$ (WLOG $a \leq b$) such that

(i) There is no pair $P = x, y$ in S_1 such that one of x, y is equal to one of a, b, but the other element of P is not larger than the other element of O.

(ii) O is not in S_2.

(iii) There are no pairs $Q = \{a', b'\}$ (WLOG $a' \leq b'$) satisfying (i) and (ii) such that $Q \neq P$, $a' \leq a$, and $b' \leq b$.

Theorem 4.3. *The (3-player) nim-pairs $N(G)$ of a game position G can be defined recursively as follows: Any terminal position has only $\{0, 0\}$ as a nim-pair. For any other position G, its nim-pairs are given by mex(S_1, S_2), where S_1 is the set of all nim-pairs of all positions that can be reached from G in exactly 1 turn, while S_2 is the set of all nim-pairs of all positions that can be reached from 0 in exactly 2 turns.*

Note that this implies, for one, a position can have more than one nim-pair.

Proof. Recall that a position is a C-position if and only if it is either terminal, or can reach another C-position in no fewer than 3 turns (which inductively means in exactly 3 turns). We use induction on the maximum possible number of steps s it can take to reach a terminal position from G (note that G is, of course, assumed to be finite), and within this also induct on the sum of the number of stones in the two nim-piles. For $s = 0$, G is a terminal position so its nim-pair is $\{0, 0\}$ as wanted. Now assume the statement is true for all $s < s_0$ and consider the case $s = s_0$. Clearly if for the pair $O = (a, b)$, (i) or (ii) from the mex definition is violated, $G + N_a + N_b$ can be moved to a position specified by S_1 or S_2 in at most 2 turns, which by induction is a C-position, so G cannot be a C-position. On the other hand, if all 3 conditions hold, then G cannot be moved to a C-position where the state of the altered G has a smaller s (in particular, the sub-base case $a + b = 0$ is proven), or a position with the same s that satisfies (iii) above; by induction these are the only C-positions that could

possibly be reached in at most 2 turns, so G can't reach any C-positions in at most 2 turns, and so must be a C-position itself. Clearly then any position violating (iii) cannot be a C-position, since it leads to a C-position by moving on the nim-piles.

So the theorem holds for $s = s_0$ as well and thus for all s. □

Given this, we can immediately analyze the 3-nim sequences for some simple subtraction games. We say that the nim-pair $N(G)$ is *well-defined* if exactly one nim-pair exists.

Proposition 4.4. *For a subtraction game $S = S\{1, 2, \cdots, n\}$, the nim-pair $N(S_k)$ for a pile of size k is well-defined and is immediately periodic* $\pmod{2n+1}$. *For $k \in \{0, 1, \cdots, 2n\}$ it is determined as follows: For $k \leq n$, $N(S_k) = \{k, k\}$. For $k = n + 2m$ with $0 < m \leq \frac{n}{2}$, $N(S_k) = \{n + m, n + m\}$. For $k = n + 2m + 1$ with $0 \leq m \leq \frac{n-1}{2}$, $N(S_k) = \{2m, 2m + 1\}$.*

Proof. We use strong induction on k. For $k \leq n$ the game S_k is identical to N_k since any move possible in Nim is allowed in the subtraction game, and vice versa. Thus $N(S_k) = N(N_k) = \{k, k\}$. Now assume the statement is true for all k less than the current value of k. First consider when $k \leq 2n$. We have two cases:

Case 1: $k = n + 2m$ for some m. Then for any $j < n + m$, S_1 includes either $\{j, j\}, \{j, j + 1\}$, or $\{j - 1, j\}$. So the minimum element of a nim-pair for k must be at least $n + m$. Checking, we see that $\{n + m, n + m\}$ works. This then implies that nothing bigger can work.

Case 2: $k = n + 2m + 1$ for some m. Then for any $j < 2m$, S_1 includes either $\{j, j + 1\}$ or $\{j - 1, j\}$, since $2m + 1 \leq n$. Also, S_2 contains $N(S_{2m}) = \{2m, 2m\}$, but S_1 does not. Thus the unique nim-pair with smallest possible sum for S_k is $\{2m, 2m + 1\}$, which works. Now, clearly, any other pair $\{a, b\}$ with $2m \leq a \leq b$ and $b \neq 2m$ would violate (iii) from the mex definition.

Now if $k > 2n$, let $k = (2n + 1) * p + q$ for some $0 \leq q \leq 2n$. By the inductive hypothesis the nim-pairs for all k before to the current k are periodic mod $2n+1$, so whatever potential nim-pairs don't work for q due to violating (i) or (ii) in the mex definition also don't work for k. We just have to check that the pairs that worked before still don't violate (i) or (ii). Clearly (ii) can't be violated since the previous occurrence of the same pair would be for $k - (2n + 1)$, which is not in S_2. For $0 \leq q \leq n$, (i) is not violated since the only possible previous nim-pair including q and an element less than q is $\{q - 1, q\}$, which, if it occurs, occurs at $(2n + 1)(p - 1) + n + q$, too far away to be in S_1. For $q > n$, the same argument as when $k \leq 2n$ shows that (i) is not violated. So our claim holds for the current value of k if $k > 2n$ as well. By induction it holds for all k. □

Proposition 4.5. *Let a subtraction game S with subtraction set $\{a_1, .., a_n\}$ have nim-pair sequence $A_0, A_1, A_2, \cdots$, where A_i is the set of nim-pairs for a pile of size i. A subtraction game kS with subtraction set $\{k * a_1, .., k * a_n\}$, for some positive integer k, has the same nim-pair sequence structure; the only distinguishing feature is that each element in the sequence for kS is repeated k consecutive times.*

Proof. Since all the allowed move sizes are divisible by k, we can never move between positions with different residues $\pmod{k}$. For each of these residues, however, the lowest position will be terminal and will have nim-pair $\{0, 0\}$; after that, the nim-pairs will follow the same sequence as for S, since the S_1 and S_2 sets for the mex are always the same in each case. Thus the entire nim-pair sequence for kS consists of each element of the sequence for S repeated k consecutive times. □

Here is a another simple but important fact that follows from Theorem 4.3.

Proposition 4.6. *For any impartial 3-player game, any position G has at least one nim-pair.*

Proof. Consider S_1, S_2, and pick a pair $O = \{a, b\}$ with a, b larger than the elements of all pairs of S_1, S_2. Clearly O satisfies (i) and (ii) since it doesn't share any elements with pairs in S_1, S_2. If it violates (iii), then there is another pair $P = \{a', b'\}$ distinct from O with $a' \leq a, b' \leq b$ that satisfies (i) and (ii). Pick such a pair P with minimal sum, so clearly it cannot violate (iii). This P is then a nim-value of G. □

5 Limitations

After conducting a week's full of research, we find much room for future work. Although we have completely solved m-player Nim, we have much more work to do as far as subtraction and other impartial games go. Using the idea of summing a game position with 2 piles of Nim, we found and proved a mex-based algorithm to find nim-pairs, and thus nim-values, for any impartial game. In the future, we look to find a simpler, equivalent method to generate nim values for impartial games, perhaps using an adaptation of our preliminary conjectures. We also hope to determine how the nim-value and nim-pair concepts relate to what happens when we sum a position with more than two piles of Nim, because our current definition of nim-pairs depends on summing an impartial game with at most 2 piles of Nim. We would like to continue our work on some of our conjectures, including the S+S+S conjecture about periodicity, and possibly using a probabilistic approach to find how often the nim-values generated by our preliminary conjectures are accurate. We also plan to explore how games behave when combined with piles of games besides Nim.

6 Appendices of Code and Results

6.1 Sums of Subtraction Games with Nim: Position and Nim-Value Generation

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Scanner;
import java.io.Serializable;
import java.util.Comparator;

public class Game3 implements Serializable
```

```
{
  ArrayList<Pile> piles;
  static ArrayList<Game3> winAs;
  static ArrayList<Game3> midBs;
  static ArrayList<Game3> loseCs;
  static ArrayList<String> mexes;
  static Integer ZERO;
  static Scanner user_input = new Scanner(System.in);
  static
  {
    winAs = new ArrayList<Game3>();
    midBs = new ArrayList<Game3>();
    loseCs = new ArrayList<Game3>();
    mexes = new ArrayList<String>();
    ZERO = new Integer(0);
    mexes.add("0");
  }
  public Game3(ArrayList<Pile> _piles)
  {
    piles = _piles;
    Collections.sort(piles, new PileComparator());
    for (int k = 0; k < piles.size();)
    {
      if (piles.get(k).equals(ZERO))
      {
        piles.remove(k);
      }
      else
      {
        k++;
      }
    }
  }
  public Game3(ArrayList<Integer> _piles, int zero)
  {
    piles = new ArrayList<Pile>();
    Collections.sort(_piles);
    for (Integer pile: _piles)
    {
      if (pile != 0)
        piles.add(new Pile(pile));
    }
  }
  public static void main(String[] args)
  {
    /*
     ArrayList<Pile> piles = new ArrayList<Pile>();
     Integer pile;
     do
     {
     System.out.print("Enter next pile, or 0 to terminate:");
     pile = Integer.valueOf(user_input.next());
     if (pile != 0)
     {
```

```
 piles.add(new Pile(pile, new ArrayList<Integer>(ruleChooser())));
 }
 }
 while (pile != 0);
 runner(piles);
 }

 public static void checkNim()
 {

 ArrayList<Pile> piles = new ArrayList<Pile>();
 ArrayList<Integer> rules = ruleChooser();
 for (int pile = 1; pile < 70; pile++)
 {
 piles.clear();
 for (int k = 0; k < 3; k++)
 {
 piles.add(new Pile(pile, new ArrayList<Integer>(rules)));
 }
 runner(piles);
 }
 }
 public static void nimFinder()
 {
 // */
ArrayList<Pile> piles = new ArrayList<Pile>();
ArrayList<Integer> rules = new ArrayList<Integer>();
Integer rule;
Integer pile;
do
{
  System.out.print("Enter next allowed move, or 0 to terminate (no moves means Nim):");
  rule = Integer.valueOf(user_input.next());
  if (rule != 0)
  {
    rules.add(rule);
  }
}
while (rule != 0);
System.out.print("Enter max pile number:");
pile = Integer.valueOf(user_input.next());
for (int p = 1; p <= pile; p++)
{
  for (int k = 0; k < 20; k++)
  {
    for (int j = k; j < 20; j++)
    {
      piles.clear();
      piles.add(new Pile(p, new ArrayList<Integer>(rules)));
      piles.add(new Pile(k));
      piles.add(new Pile(j));
      runner(piles);
    }
  }
```

```
    }
    System.out.println();
    System.out.println(mexes);
    System.out.println("Compare this to generated by the mex formula:");
    System.out.println(string3s(mexify(rules, pile)));
  }
  public static void mexify() //ArrayList<Integer>
  {
//*/
    ArrayList<Integer> rules = ruleChooser();
    System.out.print("Enter max stone number:");
    int max = Integer.parseInt(user_input.next());
    System.out.println("mex generates these nim values: " + string3s(mexify(rules, max)));
//System.out.println("for reference:")
  }
  public static ArrayList<Integer> mexify(ArrayList<Integer> rules, int max)
  {
    ArrayList<Integer> outMexes = new ArrayList<Integer>();
    outMexes.add(0);
    ArrayList<Integer> inMexes1Turn = new ArrayList<Integer>();
    ArrayList<Integer> inMexes2Turn = new ArrayList<Integer>();
    int mexType;
    String ruleRep;
    for (int k = 1; k <= max; k++)
    {
      inMexes1Turn.clear();
      inMexes2Turn.clear();
      mexType = 0;
      for (Integer rule: rules)
      {
        if (0 <= k - rule)
        {
          ruleRep = Integer.toString(outMexes.get(k - rule), 3);
          if (ruleRep.equals("0"))
            mexType |= 1;
          else if (ruleRep.contains("2"))
            mexType |= 2;
          inMexes1Turn.add(outMexes.get(k - rule));
          for (Integer rule2: rules)
          {
            if (0 <= k - rule - rule2 && k - rule - rule2 < outMexes.size())
            {
              inMexes2Turn.add(outMexes.get(k - rule - rule2));
            }
          }
        }
      }
      inMexes2Turn.addAll(inMexes1Turn);
      if (mexType == 0)
        outMexes.add(mex20(inMexes2Turn));
      else if (mexType == 1)
        outMexes.add(mex10(inMexes2Turn));
      else if (mexType <= 3)
        outMexes.add(mex(inMexes2Turn));
```

```
  }
  return outMexes;
}
public static void buildUp4(int max)
{
  ArrayList<Integer> piles = new ArrayList<Integer>();
  piles.add(1);
  piles.add(1);
  piles.add(1);
  piles.add(1);
  for (int d = 1; d <= max; d++)
  {
    piles.set(3,d);
    for (int c = 1; c <= d; c++)
    {
      piles.set(2,c);
      for (int b = 1; b <= c; b++)
      {
        piles.set(1,b);
        for (int a = 1; a <= b; a++)
        {
          piles.set(0,a);
          Game3 game = new Game3(piles, 0);
          game.organize();
        }
      }
    }
  }
  System.out.println(winAs);
  System.out.println(midBs);
  System.out.println(loseCs);
}
public static void buildUp5(int max)
{
  ArrayList<Integer> piles = new ArrayList<Integer>();
  piles.add(1);
  piles.add(1);
  piles.add(1);
  piles.add(1);
  piles.add(1);
  for (int e = 1; e <= max; e++)
  {
    piles.set(4,e);
    for (int d = 1; d <= e; d++)
    {
      piles.set(3,d);
      for (int c = 1; c <= d; c++)
      {
        piles.set(2,c);
        for (int b = 1; b <= c; b++)
        {
          piles.set(1,b);
          for (int a = 1; a <= b; a++)
          {
```

```
            piles.set(0,a);
            Game3 game = new Game3(piles, 0);
            game.organize();
          }
        }
      }
    }
    System.out.println(winAs);
    System.out.println(midBs);
    System.out.println(loseCs);
  }
  public static void runner(ArrayList<Pile> _piles)
  {
    Game3 game = (new Game3(_piles));
//System.out.println(game.position());
//System.out.println(winAs);
//System.out.println(midBs);
//System.out.println(loseCs);
//System.out.print(game.position() + ", ");
    if (game.position() == 0)
    {
      Pile pile = game.piles.remove(game.piles.size()-1);
      System.out.print(""+pile + game.piles + ", ");
//do the following if mexes are needed
      if (pile.size == mexes.size())
        mexes.add(deSum(sizeStringGet(game.piles)));
    }
  }
  public byte position()
  {
    byte bestSoFar = 2;
    int n = piles.size();
    if (n == 0)
    {
      return 0;
    }
    for (int k = 0; k < n; k++)
    {
      Pile m = piles.get(k);
      if (m.rules.size() == 0)
      {
        for (int j = 0; j < m.size; j++)
        {
          ArrayList<Pile> futurePiles = new ArrayList<Pile>(piles);
          futurePiles.set(k,new Pile(j));
          Game3 futureMove = new Game3(futurePiles);
          if (loseCs.contains(futureMove))
          {
            return 1;
          }
          else if (winAs.contains(futureMove))
          {
            bestSoFar = 1;
```

```
        }
        else if (!midBs.contains(futureMove))
        {
          byte contender;
          contender = futureMove.organize();
          if (contender == 0)
          {
            return 1;
          }
          else if (contender == 1)
          {
            bestSoFar = 1;
          }
        }
      }
    }
    else
    {
      for (int j: m.rules)
      {
        if (m.size >= j)
        {
          ArrayList<Pile> futurePiles = new ArrayList<Pile>(piles);
          futurePiles.set(k,new Pile(m.size - j, m.rules));
          Game3 futureMove = new Game3(futurePiles);
          if (loseCs.contains(futureMove))
          {
            return 1;
          }
          else if (winAs.contains(futureMove))
          {
            bestSoFar = 1;
          }
          else if (!midBs.contains(futureMove))
          {
            byte contender;
            contender = futureMove.organize();
            if (contender == 0)
            {
              return 1;
            }
            else if (contender == 1)
            {
              bestSoFar = 1;
            }
          }
        }
      }
    }
  }
  return (byte) ((bestSoFar + 1) % 3);
}
public byte organize()
{
```

```
  byte pos = position();
  if (pos == 0)
  {
    loseCs.add(new Game3(this.piles));
  }
  else if (pos == 1)
  {
    winAs.add(new Game3(this.piles));
  }
  else
  {
    midBs.add(new Game3(this.piles));
  }
  return pos;
}
public static ArrayList<String> sizeStringGet(ArrayList<Pile> piles)
{
  ArrayList<String> result = new ArrayList<String>();
  for (Pile pile:piles)
    result.add(Integer.toString(pile.size,2));
  return result;
}
public static Integer mex(ArrayList<Integer> reach)
{
  for (int k = 0; ; k++)
  {
    if (!(reach.contains(k)))
    {
      return k;
    }
  }
}
public static Integer mex10(ArrayList<Integer> reach)
{
  int value10;
  for (int k = 0; ; k++)
  {
    value10 = Integer.parseInt(Integer.toString(k,2),3);
    if (!(reach.contains(value10)))
    {
      return value10;
    }
  }
}
public static Integer mex20(ArrayList<Integer> reach)
{
  int value20;
  for (int k = 0; ; k++)
  {
    value20 = Integer.parseInt(Integer.toString(k,2).replace('1','2'),3);
    if (!(reach.contains(value20)))
    {
      return value20;
    }
```

```
  }
}
public static String trimSum(String a3, String b3)
{
  int aL = a3.length();
  int bL = b3.length();
  int n = Math.max(aL,bL);
  byte currentChar;
  String result = "";
  for (int k = 0; k<n; k++)
  {
    currentChar = 0;
    if (k < aL)
    {
      currentChar += a3.charAt(aL-k-1);
    }
    if (k < bL)
    {
      currentChar += b3.charAt(bL-k-1);
    }
    currentChar %= 3;
    result = "" + currentChar + result;
  }
  return result;
}
public static String deSum(ArrayList<String> values)
{
  String result = "0";
  for (String value: values)
  {
    result = trimSum(result, trimSum(value, value));
  }
  return result;
}
public static ArrayList<Integer> ruleChooser()
{
  ArrayList<Integer> rules = new ArrayList<Integer>();
  Integer rule;
  do
  {
    System.out.print("Enter next allowed move, or 0 to terminate (no moves means Nim):");
    rule = Integer.valueOf(user_input.next());
    if (rule != 0)
    {
      rules.add(rule);
    }
  }
  while (rule != 0);
  return rules;
}
public boolean equals(Object o)
{
  if (!(o instanceof Game3))
    return false;
```

```
    Game3 g = (Game3) o;
    return piles.equals(g.piles);
  }
  public String toString()
  {
    return piles.toString();
  }
  public static ArrayList<String> string3s(ArrayList<Integer> list)
  {
    ArrayList<String> output = new ArrayList<String>();
    for (Integer item: list)
    {
      output.add(Integer.toString(item,3));
    }
    return output;
  }
  final class PileComparator implements Comparator<Pile>
  {
    public int compare(Pile a, Pile b)
    {
      int len = Math.min(a.rules.size(), b.rules.size());
      for (int ind = 0; ind < len; ind++)
      {
        int va = a.rules.get(ind), vb = b.rules.get(ind);
        if (va != vb)
          return va > vb ? 1 : -1;
      }
      if (a.rules.size() != b.rules.size())
        return a.rules.size() > b.rules.size() ? 1 : -1;
      return a.size.compareTo(b.size);
    }

    @Override
    public boolean equals(Object o) {
      return o instanceof PileComparator;
    }
  }
}

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Scanner;
import java.io.Serializable;

public class Pile implements Serializable
{
    Integer size;
    ArrayList<Integer> rules;
    public Pile(Integer _size)
    {
        this(_size, new ArrayList<Integer>());
    }
    public Pile(Integer _size, ArrayList<Integer> _rules)
```

```
    {
        size = _size;
        rules = _rules;
        Collections.sort(rules);
    }
    @Override
    public boolean equals(Object o)
    {
        if (!(o instanceof Pile))
        {
            if (o instanceof Integer)
                return size.equals((Integer) o);
            return false;
        }
        Pile g = (Pile) o;
        return size.equals(g.size) && rules.equals(g.rules);
    }
    @Override
    public String toString()
    {
        return "" + size;// + " "+ rules;
    }
}
```

6.2 Generation of ABC positions for Single Pile Subtraction Games

```
import java.util.ArrayList;
import java.util.Scanner;

public class Nim3Test
{
  static Scanner user_input = new Scanner(System.in);
  public static void main(String[] args)
  {
    System.out.println(onePileTest(500));
  }
  public static ArrayList<Byte> onePileTest(int max)
  {
    ArrayList<Byte> alloweds = new ArrayList<Byte>();
    Byte allowed;
    do
    {
      System.out.print("Enter next allowed move, or 0 to terminate:");
      allowed = Byte.valueOf(user_input.next());
      if (allowed != 0)
      {
        alloweds.add(allowed);
      }
    }
    while (allowed != 0);
    return onePileTest(alloweds, max);
  }
  public static ArrayList<Byte> onePileTest(ArrayList<Byte> alloweds, int max)
  {
```

```
    ArrayList<Byte> singleNimNP = new ArrayList<Byte>();
    ArrayList<Byte> doubleNimABC = new ArrayList<Byte>();
    for (int k = 0; k <= max; k++)
    {
      byte currentPosS = 0;
      byte currentPosD = 0;
      for (Byte item: alloweds)
      {
        if (k >= item && singleNimNP.get(k-item) == 0)
        {
          currentPosS = 1;
          break;
        }
      }
      for (Byte item: alloweds)
      {
        if (k >= item)
        {
          if (doubleNimABC.get(k-item) == 0)
          {
            currentPosD = 1;
            break;
          }
          else if (doubleNimABC.get(k-item) == 1)
          {
            currentPosD = 2;
          }
        }
      }
      singleNimNP.add(currentPosS);
      doubleNimABC.add(currentPosD);
    }
    System.out.println(singleNimNP);
    return doubleNimABC;
  }
}
```

6.3 Mathematica Program to Find Periodicity of Combination Games

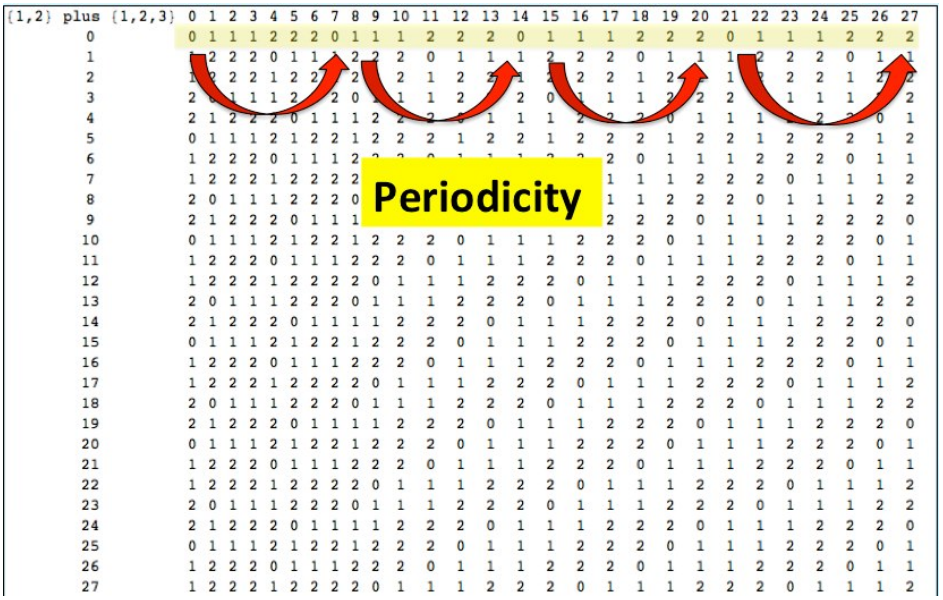

Figure 1: Periodicity of a Game with one Pile of $\{1, 2\}$ and one Pile of $\{1, 3\}$

References

[1] Weisstein, Eric W. "Nim." From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/Nim.html

[2] Loeb, Daniel E. "Stable winning coalitions." Games of No Chance 29 (1996): 451-471.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.5521&rep=rep1&type=pdf