Thesis Summary

1 Subtraction Division Games

Main Question

A subtraction–division game is a two-player combinatorial game. It starts with a total \( n \), the players take turns to decrease the total, and the winner is the first player to get to 1. At each turn, the players may either subtract \( s \in S \) from the total, or divide the total by \( d \in D \) and round up. The main problem is, given two sets \( S \) and \( D \), to determine which values of \( n \) are first player wins and which values of \( n \) are second player wins.

Some versions and variations of this have been studied by Aviezri Fraenkel, under the names of Mark and UpMark (these refer specifically to subtracting one, dividing by two, and rounding down or up). It was also a Monthly problem.

Main Result

I give a complete characterization of values of \( n \) for \( S \) and \( D \) under the following conditions: \( S = \{a\} \), \( D = \{b\} \), \( b \) even, and every prime factor of \( a \) is also a prime factor of \( b \). Moreover, under these conditions I prove that the Sprague Grundy sequence of the game (for all values of \( n \)) is \( b \)-automatic. When \( a = 1 \), I give a full set of recurrences that describe the behavior of the Sprague-Grundy sequences.

For sequences with other values of \( a \), their automaticity follows as a corollary based on a behavior called “stuttering”: the sequence eventually develops large blocks with the same Sprague-Grundy value. Because of this, we are able just to consider one value from each of the blocks, and the resulting subsequence is related to the sequence of a different game with a much smaller value of \( a \). The condition on the prime factors of \( a \) allows us to reduce to the case where \( a = 1 \).

Proof Technique

These results, particularly the recurrence relations describing the behavior of the Sprague-Grundy sequences, were obtained experimentally for \( a = 1 \) and \( b = 2 \), and proven in full generality after the fact. The proofs themselves are very simple, and are proved recursively by appealing to the definition of the game. But there is a large family of recurrences needed, and they are built up in four groups, based on the base-\( b \) representation of \( n \): whether two least significant digits are both even, both odd, or one even and one odd. We start with the simplest cases, and subsequent groups depend on the recurrences from the earlier groups.

For values of \( a \) other than 1, I prove that a certain amount of stuttering is inevitable, and characterize how the reduction of \( a \) occurs. The proof of inevitability involves showing that there are no infinite walks in a particular directed graph that encodes information about how a sequence will behave if it does not exhibit stuttering. The characterization of how the reduction works is based directly on the underlying digraph of the game.
2 Simultaneous Cops and Robbers

Main Question

We play the following game on an infinite graph: there is one robber, and several cops at various vertices. The cops are trying to catch (i.e. land on the same vertex) as the robber, while the robber is trying to escape to infinity. Initially, the cops must start at distance $N$ away from the robber, for some large $N$, and at each round the cops and the robber move simultaneously. We assume that the cops and the robber can all see the entire graph, and that the cops may coordinate their moves. Under what conditions can the cops catch the robber?

Main Result

Suppose that the infinite graph is the Cayley graph of an infinite group $G$ and set of generators $m$. If the set of generators is (algebraically) closed under conjugation, then $2|m|$ cops suffice to capture the robber for all values of $N$, and with some simple conditions on the placement of the cops within the graph. This is tight, as the minimum degree of the graph is a trivial lower bound on the number of cops needed. It is also slightly better than required, since there is no need for coordination between the cops during the game play - each one acts independently.

Proof Technique

I decompose the game into $2|m|$ sub-games, that each cop plays individually with the robber based on each of the directions (or generators) in the graph. I prove two important lemmas: if the robber takes any move each cop can move to maintain his relative position to the robber, and if the robber takes a move in a particular direction each cop can get one step closer to winning his sub-game.

To be able to prove the two lemmas above for general Cayley graphs certain conditions about the graphs must hold, and from this it’s clear that only Cayley graphs whose generating sets are closed under conjugation will admit this strategy. The proof also yields a simple characterization of which original placements are winning positions for the cops, and a bound on the length of the winning game (should the robber choose to drag out his loss).

3 Colorful Paths

Main Question

Given a graph $G$, we would like to color it with $\chi(G)$ colors, in such a way that every vertex is at the head of a path of length $\chi$ that uses each color exactly once. This is conjectured to be always true, but it is only known in certain special cases.

Main Result

I prove that the conjecture holds for cycle permutation graphs: $2n$ vertices in the form of two cycles of length $n$, connected by an arbitrary matching (i.e., a permutation of the numbers 1 through $n$). The Petersen graph is a common example of such a graph. I can also extend the proof to graphs on $2n$ vertices with one cycle of length $n$, one collection of smaller cycles, and one perfect matching connecting the two sets.

It was previously known that this is possible whenever the size of the largest clique in $G$ is equal to the chromatic number. In the case when a cyclic permutation graph has chromatic number 2, I actually prove that there is a 3-coloring for all these graphs such that every vertex is at the head of a colorful path, regardless of their chromatic number.
Proof Technique

Essentially, I give an algorithm for coloring any such graph. The foundation of the proof is a result from list coloring: if every vertex in a cycle has a list of two available colors, and there are at least two vertices with different lists, then it is possible to properly color the cycle so that every vertex has a color from its list.

The idea is to color one of the cycles first, and try to guarantee that as many vertices as possible from both cycles are already at the head of a colorful path just in the first half of the coloring. Each vertex on the other cycle then has a list of two colors available, and by the list coloring result we can extend this partial coloring to the whole graph. Much of the work is in showing that even when we cannot guarantee that all vertices are at the head of a colorful path after the first cycle is colored, we can pick a good extension of the whole coloring that will handle any leftover vertices.

4 Extensions of the Goulden–Jackson Cluster Method

Main Question

All the work in this chapter was joint with Debbie Yuster, and was published in the Journal of Difference Equations and Applications in 2010.

The Goulden–Jackson cluster method is an efficient method for computing the generating function for the number of strings of a length \( n \) in a fixed alphabet that avoid a given set of ‘bad’ words (in the same alphabet). In this context, containing a word means containing it as a contiguous sub-word. The original method can be easily modified to compute a generating function over several variables, that groups the strings by letter sets as well as length, i.e., it can distinguish between ‘aab’ and ‘abb’, based on the number of occurrences of each individual character, even though the lengths of the words are the same. Our goal was to continue to extend the method by adding variables to the generating function, so that even more information about the strings can be obtained.

Main Result

We modify the original method to compute generating functions that also take include variables for letter pairs and letter triples. For example, including letter pair variables distinguishes between the strings ‘aba’ and ‘aab’, because even though the two strings have the same length and the same set of letters, one contains the letter pair ‘ba’ and the other does not. This is helpful from the perspective of studying a randomly created string that avoids the bad words. Including variables for letter pairs and triples allows for changing the probabilities of characters based on the preceding character or characters.

The results also include a variation on the generating function with a variable that counts the number of occurrences of bad words in a string, rather than only counting the strings that avoid the bad words.

All these methods have been implemented as a Maple package, available from the following website: [www.math.rutgers.edu/~ekupin/GJ.html](http://www.math.rutgers.edu/~ekupin/GJ.html)