2021

Computational Utilities for the Game of Simplicial Nim

Nelson Penn
University of Kentucky, nape226@uky.edu
Digital Object Identifier: https://doi.org/10.13023/etd.2021.117

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Nelson Penn, Student

Dr. Mirosław Truszczyński, Major Professor

Dr. Zongming Fei, Director of Graduate Studies

COMPUTATIONAL UTILITIES FOR THE GAME OF SIMPLICIAL NIM

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the College of Engineering
at the University of Kentucky

By

Nelson Arthur Penn

Lexington, Kentucky

Co-Directors: Dr. Mirosław Truszczyński, Professor of Computer Science

and Dr. Richard Ehrenborg, Professor of Mathematics
Lexington, Kentucky

2021

ABSTRACT OF THESIS

COMPUTATIONAL UTILITIES FOR THE GAME OF SIMPLICIAL NIM

Simplicial nim games, a class of impartial games, have very interesting mathematical properties. Winning strategies on a simplicial nim game can be determined by the set of positions in the game whose Sprague-Grundy values are zero (also *zero positions*). In this work, I provide two major contributions to the study of simplicial nim games. First, I provide a modern and efficient implementation of the Sprague-Grundy function for an arbitrary simplicial complex, and discuss its performance and scope of viability. Secondly, I provide a method to find a simple mathematical expression to model that function if it exists. I show the effectiveness of this method on determining mathematical expressions that classify the set of zero positions on several simplicial nim games.

KEYWORDS: nim, simplicial complexes, optimal strategy, function interpolation

Nelson Arthur Penn

05/05/2021
Date

COMPUTATIONAL UTILITIES FOR THE GAME OF SIMPLICIAL NIM

By
Nelson Arthur Penn

Mirosław Truszczyński
Co-Director of Thesis

Richard Ehrenborg
Co-Director of Thesis

Zongming Fei
Director of Graduate Studies

05/05/2021
Date

ACKNOWLEDGEMENTS

I would like to thank my advisors, Dr. Mirosław Truszczyński and Dr. Richard Ehrenborg, for their time and guidance. Without them, I surely would have gone down one rabbit hole too many or lost sight of my goal. They encouraged me to explore my interests and to be creative; with their help, I was able to settle on a topic that is truly exciting to me.

I would also like to thank the final member of my committee, Dr. Jerzy Jaromczyk, for providing helpful suggestions for improving my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 The original game of nim

The game of nim is a classic, two-player game with interesting mathematical properties. In the game of nim, there are a certain number of stacks of stones. Two players take turns in which they pick a single stack and remove a nonzero number of stones from that stack. The player who takes the last stone wins the game.

I will present a demonstration of how the game is played.

1. Kurumi and Jeff will play a game of nim. They have agreed that Kurumi will play first. The game starts with three stacks of stones. The left stack contains 3 stones, the middle stack contains 5 stones, and the right stack contains 4 stones.

2. Kurumi removes 2 stones from the stack on the left. Now the piles contain (from left to right) 1, 5, and 4 stones, respectively.

3. Jeff removes 4 stones from the stack on the right. Now the piles contain (from left to right) 1, 5, and 0 stones, respectively.

4. Kurumi removes 4 stones from the stack in the middle. Now the piles contain (from left to right) 1, 1, and 0 stones, respectively.

5. Jeff removes 1 stone from the stack in the middle. Now the piles contain (from left to right) 1, 0, and 0 stones, respectively.

6. Kurumi takes the last stone and wins the game.

It is clear that in the above example, Kurumi won; however, the strategy is not immediately apparent. The game of nim sparks several questions. Does it matter who begins the game? Is it possible to force a win in any state? To answer these questions: if both players are playing optimally, then the winner is known before the

game starts. In this case, whether the winner will be player one or player two only depends on the initial state.

Nim is one of many games that can be classified as *impartial*. An impartial game is any game

1. that has two players
2. in which given a single game state, each player has the same moves available to them
3. that is deterministic (has no random transitions)
4. that must end in a finite amount of time

Requirement 2 means that many strategy games like chess, go, and shogi are not impartial. For example, in chess, at a given state, white can only move the white pieces, and black can only move black pieces— thus, white and black have different sets of moves. Requirement 3 rules out any games with randomness (Texas Hold'em, Blackjack, etc.) as impartial. Also, by requirement 4, we know that in an impartial game, players cannot take a "pass", or "no-op" move.

There are a great number of games that are impartial, but they are less famous than the above examples. These include Sprouts, Cram, and others. Games that are impartial tend to be easier to analyze than those that are not and often give rise to a number of interesting mathematical properties.

1.2 The Sprague-Grundy function

Analysis of nim led to the development of the Sprague-Grundy function (see papers by Sprague 1935 and Grundy [1939]). This concept is also well summarized by Ehrenborg and Steingrímmson [1996]. The Sprague-Grundy function is a function that maps game states onto non-negative integer values according to a certain recursive definition. The Sprague-Grundy function has wide implications for not only nim,

but also the entire class of impartial games. Calculating the Sprague-Grundy function for any impartial game leads to finding an optimal strategy. Before presenting the definition of this function, I will first introduce the concept of the *minimum exclusion* of a set.

Let S be a finite set of non-negative integers. This operation defined as follows:

$$\text{mex}(S) = \min\{\mathbb{Z}^{\text{nonneg}} - S\}$$

where $\mathbb{Z}^{\text{nonneg}}$ denotes the non-negative integers, $-$ denotes the set difference operation, and min denotes the minimum element of a set. Because S is finite, this is guaranteed to be defined.

Now I will introduce the Sprague-Grundy function, $\text{sg} : \text{positions} \to \mathbb{Z}^{\text{nonneg}}$. Take an arbitrary position (or game state) p on an impartial game. If no moves can be taken in state p, then $\text{sg}(p) = 0$ [1]. Otherwise,

$$\text{sg}(p) = \text{mex}\{\text{sg}(q) : \text{there is a move from state } p \text{ to } q\}$$

Let p be a position on an impartial game. I will refer to $\text{sg}(p)$ as the *Sprague-Grundy value* of p. In addition, if $\text{sg}(p) = 0$, I will refer to p as a *zero position*; similarly, if this is not the case, I will refer to it as a *non-zero position*.

The validity of this definition can be argued from the properties of impartial games. Notice that by the definition of the minimum-exclusion, this would be undefined if one could make a move from the state p to the state p. However, due to requirement 4, this is not possible. In addition, if there were randomness in the state resulting after taking a certain move, then this function could be multivalued. However, this is ruled out by 3.

[1] In fact, this base of the recursion is covered by the equation below. If there are no moves from a given state, the Sprague-Grundy value of that state is the minimum exclusion of the empty set (which is necessarily zero).

The Sprague-Grundy function has several important properties. One is that if a position is a zero position, then a player cannot make a move from that state to another zero position. A short proof (by contradiction) of this is as follows.

Proof. Assume not; that is, assume that p, q are zero positions on some impartial game, and that there exists a move from p to q. By the definition of *zero position*, $\text{sg}(p) = \text{sg}(q) = 0$. In addition, $\text{sg}(p) = \text{mex}\{S\}$ where $S = \{\text{sg}(q) : \text{there is a move from state } p \text{ to } q\}$. Because there is a move from state p to state q, then $\text{sg}(q) = 0 \in S$. By the definition of the minimum exclusion, $\text{sg}(p) \neq 0$. This is a contradiction, so the assumption must be false. □

By similar reasoning, if a position is a non-zero position, then there must be a move to a zero position.

These two properties lead to perhaps the most practical property of the Sprague-Grundy function: how to win the game. Consider an impartial game that starts on a non-zero position. On each of the first player's turns, he/she can make a move to a zero position. Then, because it is impossible to move from a zero position to another zero position, the second player is forced to move back to a non-zero position. This continues until the first player makes a move to a zero position that has no outgoing moves. At this point, the second player would lose. This must occur because the game is finite (as it is impartial). Stemming from this, it can be asserted that if an impartial game starts on a non-zero position, then the first player can force a win; otherwise, the second player can force a win.

1.3 Simplicial nim

The focus of this work is on the game of simplicial nim, first presented by Ehrenborg and Steingrímmson [1996]. Simplicial nim refers to the game of nim where the stacks of stones are placed on vertices of a simplicial complex.

A simplicial complex is a superset of vertex-edge graphs that are ubiquitous through-

out computer science and mathematics. Whereas a graph consists of a set of vertices and a set of edges, a simplical complex is made up of a set of vertices and a set of *faces*. A face is a subset of the vertex-set, which can contain any number of vertices.

To define this rigorously, a simplicial complex is defined by a vertex-set V and a set of faces, F. For each $f \in F$, $f \subseteq V$. There is an additional requirement on simplicial complexes that if $f \in F$, then every proper subset $f' \subset f$ is also an element of F. Also, the empty set $\emptyset$ must be a face.

Although graphs can only represent the skeletons of objects such as polyhedra, these objects can be fully represented as a simplicial complex. Below is a tetrahedron represented as a simplicial complex:

$$
\begin{aligned}
V &= \{a, b, c, d\} \\
F &= \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \\
&\quad \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \\
&\quad \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}
\end{aligned}
$$

It is worthy of note that the term *faces* in the context of simplicial complexes does not have the same meaning as say, the *faces* of a polyhedron. In this representation of a tetrahedron as a simplicial complex, it is apparent that the set of faces F not only contains the faces of the polyhedron, but also every edge, vertex, and the empty set [2].

The game of simplicial nim has very similar rules to the original game of nim. Piles are arranged on each vertex of the complex. Rather than being able to select from a

[2]The simplicial complexes that model other polyhedra whose geometrical faces have more than 3 nodes necessarily contain faces that are neither a geometrical face, an edge, or a vertex of the polyhedron. A cube is a good example. If one side of the cube has vertices a, b, c, d, the simplicial complex that represents this will not only have $\{a, b, c, d\}$ as a face, but will also have every combination of 3, 2, 1, and 0 of these as a face.

single pile at a time, however, a player can choose a face (other than the empty set) of the simplicial complex, and remove 1 or more stones from each of the vertices on this face.

For example, consider a game of simplicial nim taking place on the tetrahedron complex above. At a certain point in time, the stacks on vertices a, b, c, and d contain 2, 5, 7, and 4 stones, respectively. In this situation, a player could elect to remove 2 stones from the stack on vertex a, 3 stones from the stack on vertex b, and 6 stones from the stack on vertex c. This is a valid move because $\{a, b, c\}$ is a face on the complex. In fact, there are only two types of invalid moves on this complex: taking no stones, and taking at least one stone from each of the 4 nodes.

It is a simple matter to confirm that simplicial nim, like nim, is impartial. Requirements 3, 2, and 1 are true. What remains to show is that the game is finite. Because players can not add stones to the pile, and cannot make a move which removes no stones, then the number of stones in the game must be strictly decreasing. Therefore there must exist a time when there are no stones left, at which point there are no available moves and the game must end. Thus, the game is finite, satisfying requirement 4.

The Sprague-Grundy function provides the information necessary to play any impartial game optimally. This begs the question: why are impartial games still studied if the optimal strategy can be computed in every case? The answer lies in the fact that the entire game tree must be expanded in order to compute the Sprague-Grundy value for a single state, making its computation generally exponential in time and memory as the size of the game is increased. For many classes of impartial games, the set of zero positions can be determined by a simpler relationship between the numbers of stones on each pile. In some cases, the Sprague-Grundy function can be exactly modeled by a simpler function— this is true for the game of nim (discussed later).

Much of the study on impartial games is dedicated to finding such simplifications. In this work, I present a computational suite that is intended to aid researchers in finding such simplifications for new classes of simplicial nim games.

CHAPTER 2. RELATED WORK

2.1 Nim

The optimal strategy for nim was famously found by Bouton in 1901. Given a nim game with k piles of stones, Bouton defines a *safe combination* on a nim game as a position on the game where the binary sum without carry of the numbers $p_1, p_2, \ldots, p_k$ is zero. To play optimally, a player always makes moves such that his/her turn always leaves the board in a safe combination [Bouton, 1901]. This may sound familiar, and for good reason: it happens that for nim, these safe combinations exactly make up the set of zero positions in the game. Furthermore, the Sprague-Grundy value for a given state is exactly this binary sum without carry.

It is worthy of note that the "binary sum without carry" mentioned above is the same operation as the "nim-sum", "binary digital sum", or "bitwise/binary exclusive OR." Throughout this work, I use the last term to reference it.

The study of nim and its interesting properties inspired research into the properties of variations thereof.

2.2 Nim variations

2.2.1 Fibonacci nim

The game of Fibonacci nim is a version of nim played on a single pile of stones. The rules of the game are identical to that of nim, except for that a player cannot take more than twice the number of stones the other player took on the previous turn. In addition, on the first turn of the game, it is an invalid move to take all of the stones in the pile— otherwise, the game would be quite uninteresting, as it could be won within one turn.

Although it is not immediately apparent, Fibonacci game has a unique winning strategy. It has been shown that the zero positions on this game are exactly when the stack contains a Fibonacci number of stones. Consequently, the optimal strategy is

to always take a number of stones such that a Fibonacci number remains [Whinihan, 1963].

There has been further study regarding the patterns in the Sprague-Grundy values of non-zero positions as well [Larsson et al., 2016].

2.2.2 Circular nim

A circular nim game $CN(n, k)$ is a variant of nim in which some number n of piles of stones are arranged in a circle. Players can choose some number of stones from k adjacent stacks, provided that they take at least one stone from one stack. This variant of nim was formally defined and studied in [Dufour and Heubach, 2013]. Circular nim games are a special case of simplicial nim games; a circular nim game can be modeled as nim played on a simplicial complex whose faces contain every k adjacent piles and subsets thereof.

Simpler representations of the set of zero positions have been found for many cases of circular nim problems, but not yet for a general n and k.

The paper provides results for several specific values of n and k, namely

- $CN(n, 1)$
- $CN(n, n)$
- $CN(n, n-1)$
- $CN(4, 2)$
- $CN(5, 2)$
- $CN(5, 3)$
- $CN(6, 3)$
- $CN(6, 4)$
- $CN(8, 6)$

The same authors later characterized the zero positions of the previously unsolved $CN(7,4)$, by dividing the zero positions into several cases and characterizing each separately[Dufour and Heubach, 2021].

2.2.3 Simplicial nim

Ehrenborg and Steingrímmson [1996] were the first to define the problem of simplicial nim, as an extension/generalization of other nim games. Below are listed many of the major results of that work.

- Define a circuit as a set of vertices that is not a face of the graph, but where each proper subset thereof is a face. Then a simplicial graph that is a circuit is a zero position if all of the stacks have the same number of stones.

- If ϕ is an automorphism on simplicial complex Δ, that is, it maps faces to faces and non-faces to non-faces, then a position p is a zero positionon Δ if and only if $\phi(p)$ is a zero position on Δ.

- If every circuit of a simplicial graph has a point contained in no other circuit, the zero positions of the graph are completely characterized.

- For simplicial graphs that are nim-regular (these graphs have "nim-bases" which satisfy some conditions regarding their faces and subsets of vertices), zero positions can be completely described. They have special patterning in terms of powers of 2 (a common theme in the zero positions of simplicial graphs).

- The winning strategy of nim-regular graphs has been found.

- The zero positions of simplicial graphs that are binary matroids with each singleton vertex being an independent set are completely modeled.

- For certain operations which combine or augment simplicial complexes, zero positions are maintained. Operations studied include the "join" of graphs and two operations P and B which pertain to adding or replacing nodes in a complex.

Ehrenborg and Steingrímsson left several open questions, some of which were later answered by Horrocks [2010]. His paper focuses particularly on one of these open questions: When are the zero positions of simplicial complexes closed under addition?

Let $\mathfrak{C}$ be the set of circuits of Δ. Let $e(C) = [\mathbf{I}_{i \in C} \ \forall i \in V]$. Let S be the set of all positions of the form $\sum_{C \in \mathfrak{C}} a_c \cdot e(C)$.

The paper provided a few key results. A major result of the paper was that the following statements are equivalent, which sheds a great deal of light on the question above.

- The zero positions of Δ are closed under addition.
- The zero positions of Δ are exactly the set S.
- There is not any move from one position in S to another position in S.
- There does not exist any nonempty face that contains all nonzero items of an element of S.
- A property Q based on the point-circuit incidence matrix is satisfied.

The second part of the paper describes some conditions on a simplicial complex which imply the above statements are true.

Reading [1999] studied simplicial complexes of dimension 1 (classical graphs). Reading showed that graphs are nim-regular if they do not contain some specific subgraphs. "Nim-regular" was defined in the paper defined by Ehrenborg and Steingrímmson [1996] and is mentioned above. Thus, the winning strategy of these is solved. This result has very broad coverage as it enables the formulation of the winning strategy of a great number of simplicial complexes— the author mentioned that a similar technique to what he used could potentially be applied to simplicial complexes of higher dimension.

2.2.4 Hypergraph nim

A hypergraph is similar to a simplicial complex in that it is a type of higher-dimensional graph, defined over a vertex-set V and set of faces F. Like simplicial complexes, the faces of a hypergraph are subsets of the vertex set. However, hypergraphs do not have the constraint of simplicial complexes that if f is a face of the complex, then every proper subset f' of f must also be a face. Due to this, every simplicial complex is a hypergraph, but not every hypergraph is a simplicial complex.

Hypergraph nim only differs from simplicial nim in that the structure the game is played upon is a hypergraph rather than a simplicial complex. As simplicial complexes are a special case of hypergraphs, simplicial nim games are also a special case of hypergraph nim games.

One research group in particular has produced a great deal of papers regarding hypergraph nim, a superset of simplicial nim, especially studying the function (they call it the JM function after Jenkyns and Mayberry) introduced in [Jenkyns and Mayberry, 1980] . This group has found that this JM function models the Sprague-Grundy function of a large family of hypergraphs, more than in the initial findings of Jenkyns and Mayberry. They also introduce a "tetris" function T, defined as the maximum number of moves to the end of a nim game— in the case of simplicial nim, this is the sum of elements in all the piles. In many cases, the Sprague-Grundy values of a simplicial complex are simply this tetris function of that complex.

- From [Mursic, 2019]:
  - The Sprague-Grundy function can be explicitly calculated if the hypergraph is SG-decreasing, that is, if the Sprague-Grundy function can only decrease as moves are taken. In these cases, the Sprague-Grundy function matches the tetris function described above.
  - In addition to this, a certain class of hypergraph (JM hypergraphs) are introduced, whose Sprague-Grundy functions can be explicitly modeled as

the JM function mentioned above.

- From [Boros et al., 2018]
  - A hypergraph has as transversal if it has a hyperedge whose intersection with each hyperedge in the graph is non-empty. A hypergraph is minimal transversal free if it has no transversals but every subset thereof is has a transversal. If a hypergraph is JM, then it is minimal transversal free.
  - A graph (dimension 1 simplicial complex) is JM if and only if it is connected and minimal transversal free
  - A matroid is JM if and only if it is transversal free.
- From [Boros et al., 2017]
  - This paper states some conditions on hypergraphs where the Sprague-Grundy function is exactly equal to the tetris function. It also touches upon how the Sprague-Grundy function is affected by unions of games—including that if the Sprague-Grundy function of each operand is equivalent to the tetris function, then the Sprague-Grundy function of the resulting graph will also be its tetris function.
- From [Boros et al., 2020]
  - This paper analyzed how the Sprague-Grundy function of hypergraph nim is affected under an operation called the selective compound. I think, but am not completely sure, that this might be the same as the join operation from the paper by Ehrenborg and Steingrímmson. They show a closed formula for the Sprague-Grundy function of selective compounds where there are not less than 2 piles, that satisfy some other conditions.
- From [Boros et al., 2019]

– If a hypergraph is symmetric, hyperedges are at least dimension 2, and is minimal transversal free, it is JM.

The group has also studied some other, similar operations for combining graphs.

Knowing a hypergraph is JM helps characterize its Sprague-Grundy function. The JM function presented by the paper is perhaps easier to computer than the Sprague-Grundy function itself, but it is still complex (being at least NP-hard for complexes of dimension 3 or more [Mursic, 2019]).

2.2.5 Efficient computation of Sprague-Grundy function

A large portion of the work on the subject is comprised only of theoretical results regarding classes of nim games / impartial games. Some researchers have employed programmatical techniques to generate Sprague-Grundy values for a certain nim game, such as in [Dufour and Heubach, 2013], although the efficiency of this approach was not described. Furthermore, Reading used Prolog code in order to prove which vertex-edge graphs are *Nim-regular*, a property that allows the winning strategy of a simplicial nim game to be more easily found [Reading, 1999]. However, there is some work focused on the computational aspect of this as well.

Boros, et al. provided a polynomial algorithm for calculating the Sprague-Grundy function in Wythoff's game [2013]. Wythoff's game is a variant of nim played on two piles of stones; players can elect to choose some number of stones out of a single pile, or the same number of stone from both piles.

In addition, a novel method for pruning parts of the search space in the calculation of the Sprague-Grundy function was studied by Beling and Rogalski, similar to the α-β pruning method for minimax search [Beling and Rogalski, 2020]. This work has wide implications, as this pruning method applies to any impartial game.

CHAPTER 3. SOFTWARE SUPPORT

I will now introduce a suite of computational utilities for analyzing the game of simplicial nim. The two goals of the project were to develop software to aid in the study of the properties of simplicial nim, and to demonstrate its effectiveness in several practical situations (see Chapter 4). Due to the multifaceted nature of the problem, there are several "fronts" to this code.

The core functionality of this suite is the efficient calculation of the Sprague-Grundy function, which is an integral part of all other parts of the project. The game of simplicial nim is implemented so that the calculation of the Sprague-Grundy values of a large number of game states is attainable in a short time.

Another direction of the project is analyzing the Sprague-Grundy values for a game of simplicial nim. The winning strategy of any nim game or impartial game can be found by calculating the Sprague-Grundy function of the game. However, this is not efficient in a computational sense, as it requires the ability to expand the entire game tree starting at a given state. The focus of most papers on the subject is to find simpler or more familiar ways to classify whether a given state is winning or losing, especially by finding elementary functions of stack values that accomplish this. By "elementary functions", I am referring to operations or functions that would be found in a mathematical expression. Although it is helpful to calculate large numbers of Sprague-Grundy values for a certain nim game, it is nontrivial to find patterns in those vast amounts of data and discover such simple methods to divide the winning positions from the losing ones. This is often made worse by the presence of many symmetries in the graph at hand, which further obfuscate clear patterns in the data.

In order to aid in the process of finding patterns in the states and their respective Sprague-Grundy values, I designed a "function suggester," which attempts to accomplish this task in my lieu. It is built around many concepts from the subject of

compilers— in fact, it operates by searching millions of possible programs in a small assembly language for one that can differentiate a zero position in a nim game from a non-zero position.

3.1 Architecture

This section contains notes about the architecture (classes, available functions, modules) for each section of the project below.

3.1.1 Design decisions

The nature of this suite requires a platform that:

- is fast, with low computational overhead
- has a strong level of abstraction
- is very robust
- aids in catching programmer errors
- has safe concurrency

It does not require making network requests or a UI. For these reasons, Rust was selected for the project as it satisfies the constraints specified above. Rust has a heavy focus on memory safety and encouraging robust code; in addition, it can have runtimes comparable to C and C++. This makes it an excellent choice for the task at hand.

A second major design decision I made was to implement the Sprague-Grundy function specifically for the game of simplicial nim, rather than for any impartial game. The reasoning for this was that many optimizations would only be possible if the implementation of the function depends on the rules and properties of simplicial nim. Because this section of the project was so integral to the other parts of the project, I needed it to be as performant as possible, thus necessitating those optimizations.

3.1.2 The game of simplicial nim

The model and implementation of the rules of simplicial nim are split between the `Nim` class, the `NimConfiguration` class, and the `NimState` class.

`Nim` contains implementations of the game of simplicial nim and its rules. This implements the `Game` trait from the AI suite described below; that is, it implements the functions `get_moves`, `perform_move`, and `result`. In addition, this class contains the implementation of the Sprague-Grundy function for a given state, several caching tables, and a game configuration (a `NimConfiguration` struct).

The `NimConfiguration` class contains information that is necessary across an entire game of nim, including the graph definition and precalculated symmetries.

`NimState` contains information about a specific state encountered during a game of simplicial nim. These are the actual numbers of stones on each stack.

3.1.3 AI module

The AI module includes several useful traits for defining games, moves, states, and AI agents [1], as well as some implementations of classic agents such as a random agent and an MCTS agent.

Specifically, the traits defined include the following:

```
pub trait State: Clone + Hash + PartialEq + Eq + std::fmt::Display {}


pub trait Move: Clone + Hash + PartialEq + Eq + std::fmt::Display {}


pub trait Agent<S: State, M: Move>
{
    fn recommend_move(
      &mut self,
```

[1] Traits in Rust are often referred to as *interfaces* in other languages. Whereas structs define a common way to store data and basic functionality, traits define a common usage of heterogeneous objects.

```
      game: &mut Box<dyn Game<S, M>>,
      state: &S
    ) -> M;
}

pub trait Game<S: State, M: Move>
{
    // all available moves from a certain state
    fn get_moves(&mut self, state:&S) -> Vec<M>;

    // state transition function
    fn perform_move(&mut self, state:&S, m: &M) -> S;

    // reward from the game (from the previous player's
    // point of view), None if not finished
    fn result(& mut self, state:&S) -> Option<f64>;
}
```

The traits above are useful for defining a game-agnostic interface. I implemented the `State`, `Move`, and `Game` traits for simplicial nim.

3.1.4 Function suggester

The function suggester is built out mostly in a few functions, namely `suggest_function_bfs_memory` and `suggest_function_bfs_cpu`. The former is multithreaded and memory-optimized and the latter is CPU optimized.

3.2 Implementation

3.2.1 Implementation of the game of nim

The rules of simplicial nim are relatively straightforward, and are implemented when providing the `Game` interface from the AI suite. When a move is performed, the suite simply removes the appropriate number of stones from each stack selected. A move taking more stones than are in a pile or taking no stones is not allowed.

3.2.2 Computation of the Sprague-Grundy function

The Sprague-Grundy function is defined exactly as 0, if there are no moves, or the minimum exclusion of the Sprague-Grundy values of the states reachable from the current state, otherwise. This is naturally implemented recursively, which I did. The naïve implementation of the Sprague-Grundy function is extremely slow (exponential time in terms of the size of the complex for most complexes). Despite this, I optimized its operation heavily.

There were several main performance improvements I made to the calculation of the Sprague-Grundy function.

Early stops

If the complex currently being played on has certain properties, its Sprague-Grundy value may be known. One such time is if the complex is a circuit, and the number of stones on each pile are equivalent. In this case, further recursion is not necessary, as the Sprague-Grundy value is known to be zero. In addition, if there are no longer any stacks of stones connected by a face, the game is then original nim, and the Sprague-Grundy value of the state can be quickly and easily calculated by the binary *exclusive OR* of the number of stones on each stack.

Memoization

An important optimization of the Sprague-Grundy function is memoization. In the calculation of this function for a single state, there are many cases in which the same state could be recursively expanded more than once. For example, in the calculation

of the Sprague-Grundy function of some simplicial complex on 3 vertices with stack values 2, 0, and 2, the following two paths (among others) would be explored.

$$(2, 0, 2) \to (1, 0, 2) \to (1, 0, 1) \to \dots$$

$$(2, 0, 2) \to (2, 0, 1) \to (1, 0, 1) \to \dots$$

In each of these cases, the state $(1, 0, 1)$ will be recursively expanded. In this small example, the performance decrease would not be significant; however, in a large complex, exploring a single state could be very expensive. Because of this, I added the ability to memoize Sprague-Grundy values to the library. For a given `NimConfiguration`, I maintain a hash table where the keys are vectors of the stack values and the values are the Sprague-Grundy values for their respective states. This way, if a state has been recursively explored once, it will simply be retrieved from the table subsequent times.

Symmetry removal

It is common for the simplicial complexes analyzed to have symmetries, or automorphisms. As in the last example, recursively expanding states is very expensive and should be done as little as possible. To this end, I provided a way for the Sprague-Grundy function to only recursively expand a single position out of an orbit of symmetrical positions on a complex.

Let C be a simplicial complex with vertices V and faces F. For the purposes of this paper, I define a symmetry, or automorphism, as a permutation $\pi : V \to V$ which satisfies the requirement that the $\pi(G)$ is a face of C if and only if G is a face of C. These symmetries are commonly represented as a vector of unsigned integers. For example, `1 0 2` represents a symmetry π in which node 0 is mapped to node 1, node 1 is mapped to node 0, and node 2 is mapped to itself.

Researchers have extensively studied methods for finding symmetries quickly and

removing them from the search space (see papers by López-Presa et al. [2011] and Codenotti and Markov [2013]); however, removal of symmetries in a low time complexity is not a necessity for this project. In this computational suite, symmetries are calculated as a preprocessing step, because the suite is designed to help investigate specific interesting classes of simplicial complex rather than an arbitrary complex created at runtime. In addition, many of the complexes and case studies I have done are small. The limiting factor of performance in my case is not the size of the complexes— rather, it is the exponentially increasing number of positions to explore when the number of stones on each stack is increased.

Once these symmetries are calculated for a given simplicial complex, they are stored in the `NimConfiguration`. Then, whenever a Sprague-Grundy value is successfully calculated for a game state, it is added to the caching table as well as all positions which are symmetrical to it. That is, if the Sprague-Grundy value for position p is found to be k, then for every position q symmetrical to p, $q \to k$ is added to the table.

This can be illustrated through the path on two vertices. It has two symmetries, `0 1` (the identity, which I exclude from calculations) and `1 0`. Consider the state `1 2` where there is a single stone on the first stack and 2 stones on the second stack. The Sprague-Grundy value of this state is calculated as 3 and the following key-value pair is added to the caching table.

$$(1, 2) \to 3$$

In addition, the same discovered Sprague-Grundy value is added for each symmetrical position to this. In this case, there is one non-identity symmetry, `1 0`. Thus, the following will also be added to the caching table.

$$(2, 1) \to 3$$

3.2.3 Function suggester

As mentioned earlier, given a simplicial complex, it is desirable to devise a simple method to determine whether any given position on that complex is a zero position or a non-zero position.

What the previous section accomplishes is the ability for calculating Sprague-Grundy values very quickly. This enables one to calculate Sprague-Grundy values for a large number of positions on a simplicial complex. Upon visual inspection of these Sprague-Grundy values, sometimes a simple method as described above is immediately apparent; other times, it is very difficult to find any pattern in the stack values that always holds for zero positions and never holds for nonzero positions.

The function suggester is designed to address this issue computationally. Let $f(x, y, z, \ldots)$ be a function whose domain is some number of integer inputs, which outputs a single integer. The function suggester is designed to search programs of a mini-assembly language in order to find a program that models this function exactly for some finite number of inputs. That is, for every input $(x, y, z, \ldots)$ in such a finite set, the program can operate on a stack that initially contains `x, y, z, ...`, transforming it into some sequence of values `a, b, c, ...,` $f(x, y, z, \ldots)$. As the goal is to use the function suggester to generate simple mathematical expressions that fit these requirements, the mini-assembly language used contains only elementary operations—no looping or control structures. In this way, the function suggester, given a list of inputs $(x_1, y_1, z_1, \ldots), (x_2, y_2, z_2, \ldots)$ and outputs $f(x_1, y_1, z_1, \ldots), f(x_2, y_2, z_2, \ldots)$, can find a mathematical expression that can generate the correct output for each input in the list.

Since the function suggester operates on a finite set of inputs and outputs, it is guaranteed to find a solution, given enough time. The naïve solution to this problem

is a program that is equivalent to

$$g(x, y, z, \ldots) = \begin{cases} f(x_1, y_1, z_1, \ldots) & \text{if } (x == x_1 \wedge y == y_1 \wedge z == z_1 \wedge \cdots) \\ f(x_2, y_2, z_2, \ldots) & \text{if } (x == x_2 \wedge y == y_2 \wedge z == z_2 \wedge \cdots) \\ \ldots \end{cases}$$

This, of course, is not the desired answer as it correctly models the function for the finite set of inputs and outputs provided, but not others. This is not an issue in practice because the naïve solution is quite long, and all versions of the function suggester search in order of program length. Nonetheless, any program it finds is intended to be accompanied by a manual proof.

The mini-assembly language contains the following instructions. (When two values are popped off the stack, I will call value that was originally in the lower stack position the "first" and the value that was in the higher stack position the "second", assuming the stack grows upward.)

- `Plus`: pops two values off the stack, adds them, and pushes the result
- `Minus`: pops two values off the stack, subtracts the second from the first, and pushes the result
- `Mul`: pops two values off the stack, multiplies them, and pushes the result
- `Div`: pops two values off the stack, divides the first by the second (using integer division), and pushes the result
- `And`: pops two values off the stack, calculates their bitwise AND, and pushes the result
- `Or`: pops two values off the stack, calculates their bitwise OR, and pushes the result

- `Xor`: pops two values off the stack, calculates their bitwise exclusive OR, and pushes the result
- `Not`: pops a single value off the stack, calculates its bitwise NOT, and pushes the result
- `Zero`: pushes a zero onto the stack
- `One`: pushes a one onto the stack
- `LeftShift`: pops two values off the stack, bitwise left-shifts the first by the second, and pushes the result
- `RightShift`: pops two values off the stack, bitwise right-shifts the first by the second, and pushes the result
- `BooleanEq`: pops two values off the stack, compares the two values, and pushes the result (1 if the values were equal, 0 otherwise)
- `BooleanNot`: pops a single value off the stack, pushes the result (1 if the value was 0, 0 otherwise)
- `Push(id)`: pushes the number of stones for node with id `id` onto the stack

The function suggester can be used to attempt to model any function as a simple mathematical expression. However, the Sprague-Grundy function more often than not can not be represented by such an expression. For the purposes of this work, I use the function suggester to search for a program that models a slightly less complicated function: that function returns 0 if the position input is a zero position, and 1 otherwise. A search is then performed for a program that, for all of the positions collected, returns 0 if the position has a Sprague-Grundy value of 0 and 1 otherwise.

CPU-optimized BFS

This was the first successful revision of the function suggester [2]. It is stack-based breadth first search on the space of programs. It is able to effectively prune sections of the search space (not considering any program with an invalid prefix that encounters an error when running); however, its memory usage is exponential, and quickly exhausts the available RAM on the computer.

Memory-optimized BFS

This second successful revision of the function suggester was created to address the issue of the memory usage of the previous version. It explores programs according to a predefined ordering (with the ability to calculate the next program to explore from the last). This ordering is based on addition of numbers in different number bases:

Assume the mini-assembly language is comprised of instructions $x_0, x_1, ..., x_{n-1}$. (The value of n depends on the number of vertices in a given simplicial complex.) For each non-negative integer k, this version of the function suggester will explore the program $x_{i_1}, x_{i_2}, ...$ where $i_1, i_2, ...$ are the digits of k in base n, in order of least-to-most significant.

This way, the function suggester can explore a practically infinite space without exhausting the memory of the computer. In addition, this method works well with multithreading— it is embarrassingly parallel. The search space is easily divided up into a number of threads in which each thread starts with $k_0 = 0, 1, ..., t-1$, respectively and explores $k_s = k_{s-1} + t$ where t is the number of threads. Due to its parallel nature, it outperforms the CPU-optimized version considerably on a multicore computer; however, the way it is structured, it is difficult to add efficient search-space pruning to it.

[2] The first attempt was an implementation of A* with a heuristic that rates programs based on the percent of correct answers they generated. However, this created an issue of infinitely expanding a successful program with operations that did not affect the final result. The issue with this approach was most likely that the heuristic is not admissible. I leave it as an open question to find a heuristic that is appropriate for the function suggester.

CHAPTER 4. CASE STUDIES

4.1 Performance

Due to the recursive nature of the Sprague-Grundy function, the complexity of my implementation is still exponential in time. See Figures 4.1 and 4.2 for two experiments showing the runtime for calculating Sprague-Grundy values in different situations, clearing the memoization tables in between each test [1].

Despite having exponential runtime in the worst case, the application returns quickly (within 1s of runtime) for more than 20 stones per node on the P_3 complex, and for the star graph with up to 7 stacks of 4 nodes each. For the various classes of graphs studied in this work, patterns in zero positions appear within relatively small graph sizes (even being clear within 3 stones per stack on P_3 complexes, but usually within at most 7 or 8 stones per stack).

In complexes with a large number of symmetries, such as the "projective plane" complex (which has no fewer than 60 automorphisms), the implementation I provide shows its strength. The application was able to record the Sprague-Grundy value for all positions (a, b, c, d, e, f), with $0 \leq a, b, c, d, e, f < 16$ on the "projective plane" complex within 15 minutes. This is $16^6 = 16,777,216$ positions. Note that because the Sprague-Grundy value for every such position is recorded, the early stops described do not affect the runtime; however, the symmetry removal and memoization makes a heavy impact on the runtime— without it, performing the same operation took over 9 hours to complete.

Similarly, the function suggester takes exponential time in terms of the length of the minimum satisfying program (not, however in terms of graph size). See Figure 4.3 for a record of the function suggester's time-to-solution on star graphs of various sizes. The time increases exponentially, as star graphs with more leaves have increasingly

[1] All tests are run on a Linux laptop with 16 GB of RAM and a 4-core Intel 7th Gen CPU. In order to decrease variability in results, these tests are run with all other applications closed.

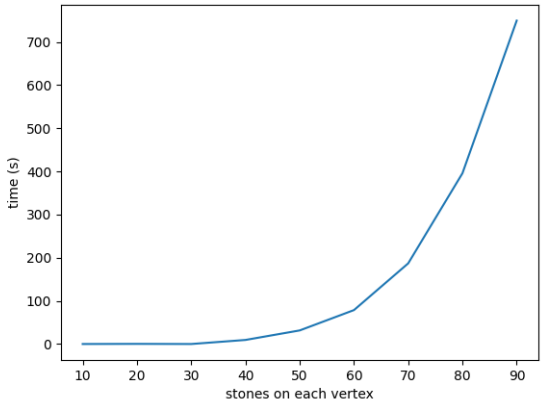

Figure 4.1: The time taken to calculate the Sprague-Grundy value of the P_3 graph, varying the number of stones in each pile.

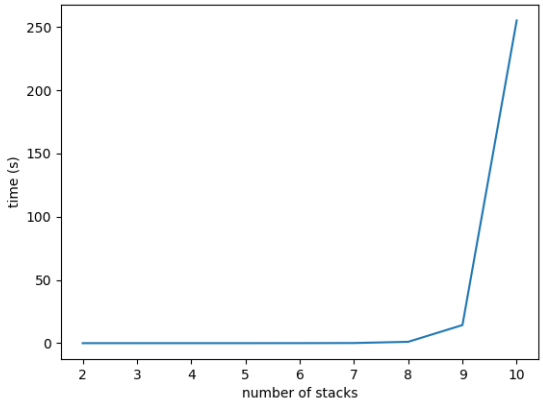

Figure 4.2: The time to calculate the Sprague-Grundy function of the star graph with 4 stones in each pile, varying the number of piles of stones.

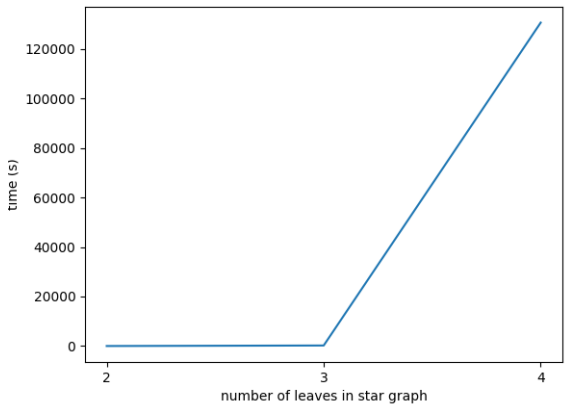

Figure 4.3: The time-to-solution of the function suggester run on the Sprague-Grundy values of star graphs, varying the number of leaves. In each case, Sprague-Grundy values for all positions that have a number of stones in the interval $[0, 6)$ in each stack were pre-generated.

longer solutions.

4.2 Playing on specific complexes

4.2.1 P_3

The path on 3 nodes is a small graph, but interesting to study. Its zero positions have a very simple form, but its Sprague-Grundy function is quite complex. A view of its fractal-like Sprague-Grundy function can be found in Figure 4.4.

After generating the Sprague-Grundy values for positions (a, b, c) on this graph where a, b, and c are anything between 0 and 7 stones, the function suggester is able to classify its zero positions with the program

```
Push(2)
Push(0)
BooleanEq
Push(1)
RightShift
BooleanNot
```

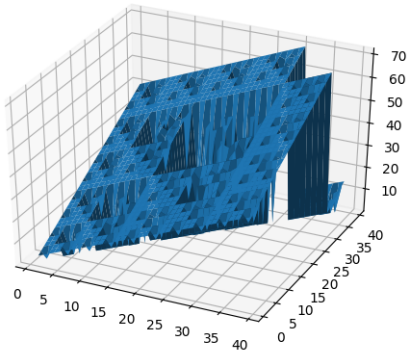

Figure 4.4: A sample of Sprague-Grundy values generated for simplicial nim played on the 3-vertex path (vertices a, b, and c), where the middle vertex, b, is 1.

I demonstrate its operation in Table 4.1. In this table, $I_{\text{condition}}$ denotes the indicator function.

To understand this result, it is helpful to examine further. First, $I_{a=c}$ will contain 1 if $a = c$, 0 otherwise. The expression $I_{a=c}>>b$ can be more easily analyzed case-by-case. See Table 4.2 for a display of the possible values of this expression.

As is apparent, this expression returns 1 if and only if $a = c$ and $b = 0$. Finally, taking the Boolean negation of this, the entire expression returns 0 if and only if $a = c$ and $b = 0$. Indeed, this corresponds to the zero positions found analytically (returning 0 for positions of the form $(a, 0, a)$ and 1 otherwise, marking the former as zero positions).

4.2.2 Star graph

The star graph is one possible generalization of P_3. This class of graphs consists of a central vertex, and radial edges to some number of leaves (See Figure 4.5). Let Star_n denote the star graph with n leaves. The zero positions of star graphs are similar to those of P_3: zero positions are exactly when the middle vertex has zero stones, and the *exclusive OR* of the outer vertices equals zero. The function suggester is also able to find a correct program to classify these: the programs it has found are

Table 4.1: Step-by-step translation of an example program in the mini-assembly language to a mathematical expression.

Step	Description	Stack
1	Load the value for face at offset 2 (c) onto the stack	c
2	Load the value for face at offset 0 (a) onto the stack	c, a
3	Pop and compare the last two values. Push 1 if equal, 0 otherwise	$I_{a=c}$
4	Load the value for face at offset 1 (b) onto the stack	$I_{a=c}, b$
5	Pop the last two values. Push the first shifted right by the second onto the stack.	$I_{a=c}>>b$
6	Pop the last value. Push the Boolean negation of the result.	$\neg(I_{a=c}>>b)$

Table 4.2: In-depth analysis of the expression $I_{a=c}>>b$.

$I_{a=c}$ \ b	0	nonzero
0	0	0
1	1	0

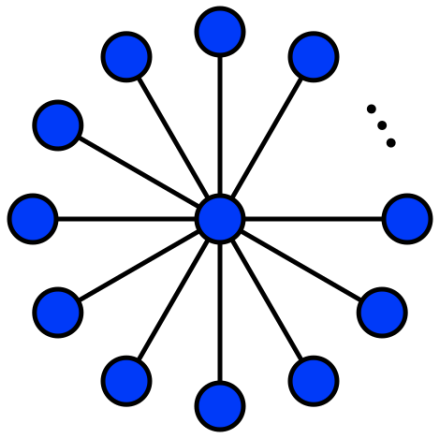

Figure 4.5: A generalized star graph

listed below. In each of the cases, the vertex at the center of the star graph has id 0.

For the star graph with 2 leaves (equivalent to P_3):

```
Push(2)
Push(1)
BooleanEq
Push(0)
RightShift
BooleanNot
```

For the star graph with 3 leaves:

```
Push(2)
Push(3)
Push(1)
Xor
BooleanEq
Push(0)
RightShift
BooleanNot
```

For the star graph with 4 leaves:

```
Push(4)
Push(3)
Push(2)
Push(1)
Xor
Xor
```

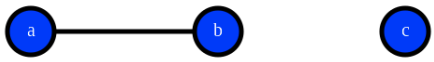

Figure 4.6: The "shriek" graph

```
BooleanEq
Push(0)
RightShift
BooleanNot
```

It began to become infeasible to calculate such programs for star graphs with 5 leaves or more in terms of time required (see Figure 4.3).

4.2.3 "Shriek" graph

The shriek graph is an interesting case: it is one of the few dimension-1 graphs that Reading found to have no Nim-basis [Reading, 1999]. However, its set of zero positions can still be easily characterized.

The shriek graph contains two vertices connected by a single edge and one node that has degree of zero (no outgoing or incoming edges). See Figure 4.6 for a visual representation of this graph. Its zero positions take the form $(a, b, a + b)$ where the first two vertices are those joined by an edge, and the third is the vertex with degree zero. This is one of the special cases in which the function suggester was not only able to create a function to model the zero position of the complex, but also model the Sprague-Grundy function itself:

```
Push(2)
Push(1)
Push(0)
Plus
Xor
```

It is straightforward to confirm that this output matches the Sprague-Grundy function of the shriek graph, which is $\text{sg}(p) = (a + b) \oplus c$, where position p is the tuple (a, b, c) with the vertices ordered as above.

4.2.4 Circular nim

I also studied the effectiveness of the function suggester in the domain of circular nim games, which are defined in Section 2.2.2.

I tested my function suggester on many of the small circular nim games. In a short time (over the span of a few hours), it was able to rediscover the characterizations of the zero positions for the following complexes that the authors solved in their original paper: $CN(2, 1)$, $CN(2, 2)$, $CN(3, 1)$, $CN(3, 2)$, $CN(3, 3)$, $CN(4, 1)$, and $CN(4, 2)$.

I have confirmed that the programs generated correspond to the theoretical results proven by Dufour and Heubach [2013]; however, the solutions are sometimes unintuitive[2]. For brevity, the satisfying programs are not listed here. See Appendix 5.2 for the full output.

I expect that, given more time, it could also solve at least $CN(4, 3)$, $CN(4, 4)$, $CN(5, 1)$, and $CN(5, 5)$.

The smallest case for which the authors did not provide a solution was $CN(6, 2)$. I generated the Sprague-Grundy values for a subset of the positions on this game on

[2]For example, using `Mul` on Boolean values for what would usually be the nor operation, or using `..., BooleanNot, BooleanNot` to produce as the operation ... == 1. The results also make it clear that the multithreaded version does not guarantee finding a minimal program, as one program uses `..., Mul, BooleanNot` rather than `..., And`.

which I then tested the function suggester. However, this did not terminate in any reasonable time, thus not providing a solution.

It is worthy of note that $CN(6,2)$ is the smallest circular nim complex to contain the vertex-induced subgraph of P_3 with an additional disjoint vertex, which I will call the "P_3 plus one" complex. This subcomplex is quite difficult; in fact, neither I nor the function suggester has been able to characterize its zero positions. This is because a position on this complex is a zero position if and only if the Sprague-Grundy value of the P_3 subcomplex is equal to that of number of stones on the free vertex. The implications of this are that characterizing the zero position of the "P_3 plus one" complex is equally as difficult as characterizing the Sprague-Grundy function of P_3, which is nontrivial. Furthermore, we can say that characterizing the zero positions of $CN(6,2)$ is at least as hard as characterizing the Sprague-Grundy function of P_3.

The difficulty of the "P_3 plus one" subcomplex also prevented the successful analysis of multiple copies of P_3 and many other simplicial complexes. This fact, together with suggestion from my advisor, led to my study of the "projective plane" complex below, as it does not have "P_3 plus one" as any vertex-induced subcomplex.

4.2.5 "Projective plane" complex

The "projective plane" complex is a simplicial complex on 6 nodes whose faces include the empty set, all six nodes, every pair of nodes (that is, every non-loopback edge), and 10 faces of cardinality 3. The following is its formal definition:

The "projective plane" complex is a simplicial complex over vertex-set V and face-

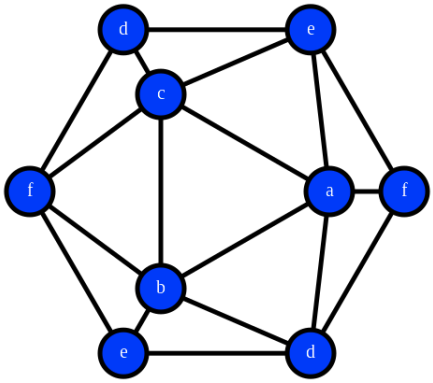

Figure 4.7: A graphical depiction of the "projective plane" complex. In this image, each vertex, edge, and triangle is a face. It gets its nickname because opposite vertices are considered the same, thus having wraparound similar to the projective plane of real numbers.

set F where

$$
\begin{aligned}
V &= \{a, b, c, d, e, f\} \\
F &= \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \\
&\quad \{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{a, f\}, \\
&\quad \{b, c\}, \{b, d\}, \{b, e\}, \{b, f\}, \\
&\quad \{c, d\}, \{c, e\}, \{c, f\}, \\
&\quad \{d, e\}, \{d, f\}, \{e, f\}, \\
&\quad \{a, b, c\}, \{a, b, d\}, \{a, c, e\}, \{a, d, f\}, \{a, e, f\}, \\
&\quad \{b, c, f\}, \{b, d, e\}, \{b, e, f\}, \\
&\quad \{c, d, e\}, \{c, d, f\}\}
\end{aligned}
$$

Despite being promising in terms of never having "P_3 plus one" as a vertex-induced subcomplex, the "projective plane" complex still defies characterization of its zero positions. Despite allowing the function suggester to run for days and inspecting

the Sprague-Grundy values in detail myself, I was not able to find any one unifying pattern of the zero positions.

I was only able to characterize some (largely trivial) subsets thereof. The zero positions of the "projective plane" complex include all symmetries of $(0, 0, 0, a, a, a)$ and $(0, 0, a, b, a + b, a + b)$.

There are some classes of positions that were very frequently occuring amongst the set of zero positions, but I haven't yet completely characterized. I leave them as open questions: For what constraints on a, b, and c are positions (a, a, b, b, c, c) zero positions? Also, for what constraints on a, b, and c are positions $(a, a, b, c, a + b + c, a + b + c)$ zero positions?

CHAPTER 5. CONCLUSION

The game of simplicial nim can be very difficult to analyze, computationally or analytically. However, this work has made several contributions to its study.

One contribution is a modern and efficient implementation of the calculation of the Sprague-Grundy function for a simplicial nim game. This is sped up by the inclusion of several theoretical results, memoization, and symmetry removal.

Also, I designed a "function suggester," which is a computational method that suggests a simple elementary relation to model a function. I used this to search for programs that determine whether a position on a game of simplicial nim is a zero position or a non-zero position, and demonstrated its effectiveness in rediscovering known patterns in the zero positions of several classes of simplicial complexes.

At the inception of the project, the suite was only capable of calculating the Sprague-Grundy function of very small complexes. After many optimizations, the calculation of the Sprague-Grundy function has become quite effective. At the time of writing this, on modest hardware (a computer with 16 GB of RAM and a 4-core Intel i7 CPU), the suite was able to calculate over 16 million Sprague-Grundy values on the "projective plane" complex in around 15 minutes.

Furthermore, the function suggester has been proven to be capable of generating appropriate programs to classify positions on simple simplicial complexes as zero positions or non-zero positions.

5.1 Limitations

One of major the limitations of the project is the capability of the function suggester. Its purpose was to provide some computational insight into finding simple patterns that split zero positions from non-zero position)s in a simplicial complex. As shown in the case studies, it is successfully able to solve several simple complexes; however, it does not yet seem to beat visual inspection. Consider star graphs, for

example. By simply inspecting the Sprague-Grundy values for a large number of zero positions, it is easy to see that the middle vertex is always zero. From there, it is quite clear that the remaining nodes are the original game of nim. This would naturally lead to the (correct) hypothesis that a position on a star graph is a zero position if and only if the middle vertex has 0 stones and the *exclusive OR* of the remaining stack values equals 0. The function suggester is also able to come to this conclusion, but depending on the size of the star graph in question it takes anywhere from a few minutes to many hours to discover this.

For more complicated graphs, where the zero positions do not have an immediately clear, unifying pattern, the function suggester does no better than the intuition of the researcher. This has manifested itself in the case studies treating $CN(6, 2)$ as well as the "projective plane" complex.

5.2 Future directions

There are several open ends to this work: further optimizing the calculation of the Sprague-Grundy function, improving the function suggester in order to make it surpass human ability, solving and proving winning strategies on the projective plane complex, and allowing the function suggester to produce a program to classify zero positions without respect to symmetries.

The first direction is improving the calculation of the Sprague-Grundy function further. I would like to see if some of the concepts from the search-space pruning method presented in [Beling and Rogalski, 2020] can be incorporated into the calculation of the Sprague-Grundy function. I believe that this would be nontrivial and may exclude the possibility of some of the other optimizations I made, such as symmetry removal; however, perhaps it could be incorporated if I modified the method these optimizations are performed. However, I was not able to explore this due to time constraints.

Second, the function suggester's capabilities could be improved. As mentioned in

the limitations section, it does not seem to be more powerful than informed inspection. This could be improved in a few ways. One is adding effective search space pruning to the memory-optimized version, as was added to the CPU-optimized version. I attempted to do this with a trie that contains disallowed program prefixes (these disallowed prefixes are invalid programs, which will always exhaust the program's stack and error). This was shared across threads. However, the time complexity of looking up the prefix of a program in the trie was the same as running the program to see if an error occurs. That and the additional overhead from communicating between threads caused this disallowed prefix trie to slow down, rather than speed up, the search. Despite this, I still believe that there is opportunity in pruning the search space. This aspect could be further extended if an effective method of pruning programs that are functionally identical due to the commutativity of operations (e.g. `One; Zero; Plus;` vs. `Zero; One; Plus;`, etc.).

The function suggester could also be improved if an appropriate heuristic for the search could be found. The best version of the suggester that I created is pure breadth-first search— it is guaranteed, if run on a single thread, to find the shortest program that classifies zero positions for all the calculated positions on the game. However, for simplicial complexes where the number of instructions in such a program is expected to be high, the search will take a very long time as its runtime is exponential in the length of the solution. Because of this, it could benefit from search using a heuristic that is both correct and approaches the solution faster that pure breadth-first search. I was unable to devise such a heuristic— it is an interesting open question for future research.

One outstanding mystery is the set of zero positions on the projective plane simplicial complex. It had promise as it does not have P_3 as a vertex-induced subcomplex; however, I have not been able to find any pattern that can classify a general position on this complex as a zero position or non-zero position, either through the function

suggester or my own intuition. It is possible that its zero positions positions cannot be classified by any elementary functions, but I have not been able to prove or disprove this thus far.

One more open question is if it is possible to factor in symmetry removal to the function suggester. I posit that it is, and provide a potential solution.

To illustrate this, I will first give a concrete example from the work of Dufour et al. [2013]. The researchers found that the zero positions of the circular nim game with 6 stacks in which a player can select stones from any 3 adjacent stacks are of the form (a, b, c, d, e, f), where $a + b = d + e$ and $b + c = e + f$. This describes zero positions for a single automorphism of the complex; that is, this form of positions and all symmetrical forms thereof, describe all zero positions for the game.

In my experiments with the function suggester, for a simplicial nim game like this, I would run the function suggester against all positions (a, b, c, d, e, f). Thus, in order for it to succeed, it would have to create a program that classifies all symmetrical permutations of this:

- (a, b, c, d, e, f), where $a + b = d + e$ and $b + c = e + f$
- (a, b, c, d, e, f), where $b + c = e + f$ and $c + d = f + a$
- (a, b, c, d, e, f), where $c + d = f + a$ and $d + e = a + b$

Without symmetry removal, the function suggester would have to produce a program with about 3 times the instructions it would need with it. Take a graph with k automorphisms. Say that the shortest program that classifies zero positions for a single automorphism has n instructions. The function suggester would take $O(a^n)$ time to classifying the zero positions for a single automorphism as opposed to $O((ka)^n) = O(k^n a^n)$ time to classify the zero positions for all automorphisms. When the number of symmetries increases, the time complexity for calculating isomorphism increases extremely quickly.

If automorphisms were somehow excluded from the subset of positions that the function suggester analyzes, this would greatly boost its effectiveness, especially on complexes with many symmetries (such as the "projective plane" complex, which has 60).

I present a possible solution to this issue. I created a third implementation of the function suggester, `suggest_function_bfs_memory_wrt_symmetries`, which attempts to generate a program such that:

- For each non-zero position p, the evaluation of the program is nonzero.
- For each orbit of symmetrical positions, the evaluation of the program is zero for at least one position out of the orbit.

Due to time constraints, I wasn't able to fully evaluate the effectiveness or correctness of this approach— thus, it is not included in the body of the paper. This approach is initially promising, as it was able to replicate results for several of the smaller case studies. Furthermore, the evaluation criterion in this version begins to sound very much like an instance of the satisfiability problem. In fact, this can be encoded by a simple CNF theory in propositional logic. Is there a possibility of incorporating a well-known SAT solver to the evaluation programs in this version of the function suggester?

APPENDICES

Results of tests

Performance-related information

Raw output for the test described in Figure 4.2:

```
For 2 stacks, found Sprague-Grundy value 8 in 53.264 microseconds.
For 3 stacks, found Sprague-Grundy value 12 in 435.562 microseconds.
For 4 stacks, found Sprague-Grundy value 8 in 1.486921ms.
For 5 stacks, found Sprague-Grundy value 12 in 3.874688ms.
For 6 stacks, found Sprague-Grundy value 8 in 17.50395ms.
For 7 stacks, found Sprague-Grundy value 12 in 101.676076ms.
For 8 stacks, found Sprague-Grundy value 8 in 1.06046286s.
For 9 stacks, found Sprague-Grundy value 12 in 14.297733449s.
For 10 stacks, found Sprague-Grundy value 8 in 255.126076586s.
```

Raw output for the test described in Figure 4.1:

```
For depth 10, found Sprague-Grundy value 30 in 16.625585ms.
For depth 20, found Sprague-Grundy value 60 in 319.191136ms.
For depth 30, found Sprague-Grundy value 90 in 2.169919397s.
For depth 40, found Sprague-Grundy value 120 in 9.419531734s.
For depth 50, found Sprague-Grundy value 150 in 31.489047685s.
For depth 60, found Sprague-Grundy value 180 in 78.609425394s.
For depth 70, found Sprague-Grundy value 210 in 186.935767715s.
For depth 80, found Sprague-Grundy value 240 in 395.864879494s.
For depth 90, found Sprague-Grundy value 270 in 749.705852118s.
```

Below is the raw output for the test described in Figure 4.3. In addition, this contains the satisfying programs found for the star graphs of the sizes described.

```
star graph with 2 leaves:
       found solution in 516.826917ms
       [Push(2), Push(1), BooleanEq, Push(0), RightShift, BooleanNot]

star graph with 3 leaves:
       found solution in 208.429786682s
       [Push(2), Push(3), Push(1), Xor, BooleanEq, Push(0), RightShift,
          ↪ BooleanNot]

star graph with 4 leaves:
       found solution in 130724.644186851s
       [Push(4), Push(3), Push(2), Push(1), Xor, Xor, BooleanEq, Push(0),
          ↪ RightShift, BooleanNot]
```

P_3

The function suggester found the following program to model the zero-position function of P_3, testing over all positions with less than 8 stones on each stack.

```
Push(2)
```

```
Push(0)
BooleanEq
Push(1)
RightShift
BooleanNot
```

"Shriek" graph

The function suggester found the following program for the Sprague-Grundy function of "shriek" graphs, testing over all positions with less than 8 stones on each stack.

```
Push(2)
Push(1)
Push(0)
Plus
Xor
```

Circular nim

The solutions for $CN(n, k)$ as described in Chapter 4 are shown below. Note that the timings for these were not recorded in a controlled environment, so they serve merely as estimates.

```
CN(2, 1):
    found solution in 5.384072ms
    [Push(0), Push(1), BooleanEq, BooleanNot]

CN(2, 2):
    found solution in 25.377171ms
    [Push(1), Push(0), Plus, BooleanNot, BooleanNot]

CN(3, 1):
    found solution in 683.270995ms
    [Push(2), Push(1), Push(0), Xor, BooleanEq, BooleanNot]

CN(3, 2):
    found solution in 193.510696569s
    [Push(2), Push(0), BooleanEq, Push(1), Push(0), BooleanEq, Mul,
        BooleanNot]

CN(3, 3):
    found solution in 14.009247648s
    [Push(2), Push(1), Push(0), Or, Or, BooleanNot, BooleanNot]

CN(4, 1):
    found solution in 330.871807662s
    [Push(0), Push(3), Push(2), Push(1), Xor, Xor, BooleanEq,
        BooleanNot]
```

```
CN(4, 2):
    found solution in 291.850208903s
    [Push(1), Push(3), BooleanEq, Push(2), Push(0), BooleanEq, Mul,
        BooleanNot]
```

Source code

This section contains abbreviated source code for the project. Several modules that are not critical to the work or described in the body of the paper (including several unit tests and halted directions) have been removed for brevity.

Many of the items marked as tests contain the `#[ignore]` macro. These tests are meant to be run individually rather than as unit tests— they each perform a single use-case of the library. They can be invoked by commenting out the ignore macro and then explicitly running the test using the `cargo` utility.

On a high level:

- `src/lib.rs` is the crate root, and contains functions to record the Sprague-Grundy values of many positions on a nim game.
- `src/nim/mod.rs` contains the main implementation of Nim, the Sprague-Grundy function, symmetry generation, and the like.
- `src/ai/*` contains the code related to the AI module described in the body of the paper
- `src/helpers.rs` and `src/combinator.rs` contain useful helper functions which are used throughout the rest of the project. In addition, `src/helpers.rs` contains the code for the ordering of programs used in the memory-optimized version of the function suggester.
- `src/function_suggester.rs` contains the implementation and tests of the function suggester.

Directory contents

- `Cargo.toml`
- `src/lib.rs`
- `src/nim/mod.rs`
- `src/interesting_complexes.rs`
- `src/helpers.rs`
- `src/function_suggester.rs`
- `src/combinator.rs`
- `src/ai/mod.rs`
- `src/ai/agent.rs`
- `src/ai/game.rs`

- src/ai/game_tree.rs
- src/ai/mcts_agent.rs
- src/ai/random_agent.rs
- src/ai/game_runner.rs

From Cargo.toml

```
[package]
name = "simplicial_nim"
version = "0.1.0"
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
    ↪ reference/manifest.html

[dependencies]
rand = "0.7"
chrono = "0.4.19"
```

From src/lib.rs

```
pub mod nim;
pub mod ai;
pub mod interesting_complexes;
pub mod combinator;
pub mod helpers;
pub mod function_suggester;

use nim::*;
use std::io::{BufWriter, Write};

struct InnerRecordDataArgs<'a>
{
    nim: &'a mut Nim,
    complex: &'a mut nim::NimState,
    initial_stacks: &'a Vec<u32>,
    index: usize,
    all_sg: &'a mut BufWriter<std::fs::File>,
    all_0: &'a mut BufWriter<std::fs::File>,
    symmetries_removed_sg: &'a mut BufWriter<std::fs::File>,
    symmetries_removed_0: &'a mut BufWriter<std::fs::File>
}

/// `record_data` takes a nim game, an initial state, and a name
/// and then records the Sprague-Grundy value for all states
/// where the number of stones on each stack is less than
/// the number of stones on the stack in the initial state provided.
```

```
///
/// ## Examples
/// The following code records the Sprague-Grundy values for all states (a,
    ↪  b, c) on a
/// shriek graph where a, b, c < 8 under the directory "results/shriek".
/// ```rust
/// let depth = 8;
/// let (nim, nim_state) = crate::interesting_complexes::shriek(depth,
    ↪ depth, depth);
/// record_data(nim, nim_state, "shriek");
/// ```
pub fn record_data(mut nim: Nim, mut initial_state: NimState, name: &str)
{
    use std::fs;
    use helpers::overwrite_file;

    let dir_name = String::from(
        format!("results/data/{}", name)
    );
    let all_dir = dir_name.clone() + "/all";
    let symmetries_removed_dir = dir_name.clone() + "/symmetries_removed";

    fs::create_dir_all(&all_dir).unwrap();
    fs::create_dir_all(&symmetries_removed_dir).unwrap();

    let mut all_sg = overwrite_file(format!("{}/sprague_grundy", all_dir));
    let mut all_0 = overwrite_file(format!("{}/zero_positions", all_dir));
    let mut symmetries_removed_sg = overwrite_file( format!("{}/
        ↪ sprague_grundy", symmetries_removed_dir) );
    let mut symmetries_removed_0 = overwrite_file( format!("{}/
        ↪ zero_positions", symmetries_removed_dir) );

    let initial_stacks = initial_state.stacks.clone();

    let args = InnerRecordDataArgs
    {
        nim: &mut nim,
        complex: &mut initial_state,
        initial_stacks: &initial_stacks,
        index: 0,
        all_sg: &mut all_sg,
        all_0: &mut all_0,
        symmetries_removed_sg: &mut symmetries_removed_sg,
        symmetries_removed_0: &mut symmetries_removed_0
    };

    inner_record_data(args);
```

```
    // Explicit flushing probably not necessary but helpful if there are
        ↪ issues.
    all_sg.flush().unwrap();
    all_0.flush().unwrap();
    symmetries_removed_sg.flush().unwrap();
    symmetries_removed_0.flush().unwrap();

}

fn inner_record_data(mut args: InnerRecordDataArgs)
{
    // base case: index at end
    if args.index == args.initial_stacks.len()
    {
        let _sg = args.nim.sprague_grundy_value(&args.complex);
        let found_value = args.nim.mem.lookup_table.get(&args.complex.stacks
            ↪ ).unwrap();

        let position_string = format!("{}", args.complex);

        let position_string_with_value = format!("{}\t{}", position_string,
            ↪ found_value.sg_value);

        writeln!(args.all_sg, "{}", position_string_with_value).unwrap();
        if found_value.sg_value == 0
        {
            writeln!(args.all_0, "{}", position_string).unwrap();
        }



        if found_value.original
        {
            writeln!(args.symmetries_removed_sg, "{}",
                ↪ position_string_with_value).unwrap();

            if found_value.sg_value == 0
            {
                writeln!(args.symmetries_removed_0, "{}", position_string).
                    ↪ unwrap();
            }
        }

        return;
    }
```

```
    // recursive step: permute with future indices

    for i in 0..args.initial_stacks[args.index]
    {
        args.complex.stacks[args.index] = i; // set current value

        inner_record_data(InnerRecordDataArgs{
            nim: &mut args.nim,
            complex: &mut args.complex,
            initial_stacks: &args.initial_stacks,
            index: args.index.clone() + 1,
            all_sg: &mut args.all_sg,
            all_0: &mut args.all_0,
            symmetries_removed_sg: &mut args.symmetries_removed_sg,
            symmetries_removed_0: &mut args.symmetries_removed_0

        });
    }

}


#[cfg(test)]
mod tests{
    use super::*;
    use std::fs;
    use std::io::Write;


    #[test]
    fn generate_projective_plane_symmetries()
    {
        let (nim, _nim_state) = interesting_complexes::projective_plane(3,
            ↪ 3, 3, 3, 3, 3);
        let symmetries = nim.configuration.generate_symmetries();
        assert_eq!(symmetries.len(), 59); // 60 - identity

        let mut file = fs::OpenOptions::new().write(true).truncate(true).
            ↪ create(true).open("results/data/projective_plane_symmetries")
            ↪ .unwrap();

        for symmetry in symmetries
        {
            for id in symmetry
            {
                write!(file, "{}\t", id).unwrap();
            }
```

```
        writeln!(file, "").unwrap();
    }

}

#[test]
fn generate_square_symmetries()
{
    // let nim_state = interesting_complexes::quadruped(3, 3, 3, 3, 3,
        ↪ 3, 3);
    let (nim, _nim_state) = interesting_complexes::square(3, 3, 3, 3);
    let symmetries = nim.configuration.generate_symmetries();
    for symmetry in symmetries.iter()
    {
        println!("{:?}", symmetry);
    }
    assert_eq!(symmetries.len(), 7);
}

#[test]
fn generate_path_3_symmetries()
{
    let (nim, _nim_state) = interesting_complexes::path_3(3, 3, 3);
    let symmetries = nim.configuration.generate_symmetries();
    assert_eq!(symmetries, vec![vec![2, 1, 0]])
}

#[test]
fn record_projective_plane()
{
    let depth = 16;
    let (nim, nim_state) = interesting_complexes::projective_plane(depth
        ↪ , depth, depth, depth, depth, depth);
    record_data(nim, nim_state, "projective_plane_16");
}

// #[ignore]
#[test]
fn record_shriek()
{
    let depth = 8;
    let (nim, nim_state) = interesting_complexes::shriek(depth, depth,
        ↪ depth);
    record_data(nim, nim_state, "shriek");
}

#[ignore]
```

```
#[test]
fn record_path_3()
{
    let depth = 8;
    let (nim, nim_state) = interesting_complexes::path_3(depth, depth,
        ↪ depth);
    record_data(nim, nim_state, "path_3");
}

#[ignore]
#[test]
fn record_circular_nim()
{
    let nks = vec![
        (2, 1),
        (2, 2),
        (3, 1),
        (3, 2),
        (3, 3),
        (4, 1),
        (4, 2),
        (4, 3),
        (4, 4)
    ];

    let depth = 6;

    for (n, k) in nks
    {
        let (nim, nim_state) = interesting_complexes::circular_nim(n, k,
            ↪  vec![depth; n]);
        // println!("{:?}", nim_state);
        // println!("{:#?}", nim.configuration);
        record_data(nim, nim_state, &format!("circular_nim/{}_{}", n, k)
            ↪ [..]);

    }

}

#[ignore]
#[test]
fn record_joined_path_3_graphs()
{
    let max_num_graphs = 3;
    let starting_value = 6;
```

```
    for num_graphs in 1..max_num_graphs
    {
        println!("recording data for {} copies of path_3", num_graphs);
        let (nim, nim_state) = interesting_complexes::
            ↪ joined_path_3_graphs(num_graphs, starting_value);
        record_data(nim, nim_state, &format!("joined_path_3_graphs/{}",
            ↪ num_graphs)[..]);
    }
}

#[ignore]
#[test]
fn record_path_3_and_dots()
{
    let max_extra_dots = 4;
    let starting_value = 6;

    for num_extra_dots in 0..max_extra_dots
    {
        println!("recording path_3 with {} extra dots", num_extra_dots);
        let (nim, nim_state) = interesting_complexes::path_3_and_dots(
            ↪ num_extra_dots, starting_value);
        record_data(nim, nim_state, &format!("path_3_with_dots/{}",
            ↪ num_extra_dots)[..]);
    }
}

#[ignore]
#[test]
fn record_star_graphs()
{
    let max_star_graph_size = 8;
    let starting_value = 6;

    for star_graph_size in 0..max_star_graph_size
    {
        println!("recording star with {} leaves", star_graph_size);
        let (nim, nim_state) = interesting_complexes::star_graph(
            ↪ star_graph_size, starting_value);
        record_data(nim, nim_state, &format!("star/{}", star_graph_size)
            ↪ [..]);
    }
}

#[ignore]
#[test]
fn increase_stones()
```

```
{
    use std::time::Instant;
    let mut results_file = crate::helpers::overwrite_file(String::from("
        ↪ results/compiled/increase_stones"));

    for depth in (10..100).step_by(10)
    {
        println!("Testing depth: {}", depth);
        let (mut nim, nim_state) = interesting_complexes::path_3(depth,
            ↪ depth, depth);

        let start_time = Instant::now();
        let sg_value = nim.sprague_grundy_value(&nim_state);
        let end_time = Instant::now();

        writeln!(results_file,
            "For depth {}, found Sprague-Grundy value {} in {:?}.",
            depth,
            sg_value,
            end_time - start_time
        ).unwrap();

        results_file.flush().unwrap();
    }
}

#[ignore]
#[test]
fn increase_stacks()
{
    use std::time::Instant;
    let mut results_file = crate::helpers::overwrite_file(String::from("
        ↪ results/compiled/increase_stacks"));

    let depth = 4;

    for num_leaves in 1..20
    {
        println!("Testing {} leaves", num_leaves);
        let (mut nim, nim_state) = interesting_complexes::star_graph(
            ↪ num_leaves, depth);

        let start_time = Instant::now();
        let sg_value = nim.sprague_grundy_value(&nim_state);
        let end_time = Instant::now();

        writeln!(results_file,
```

```
                "For {} stacks, found Sprague-Grundy value {} in {:?}.",
                num_leaves + 1,
                sg_value,
                end_time - start_time
            ).unwrap();

            results_file.flush().unwrap();
        }
    }
}
```

From src/nim/mod.rs

```
use crate::ai;
use ai::game as game;
use ai::game::*;
use std::collections::{HashSet, HashMap};
use crate::combinator::Combinator;
use crate::intersperse_for;


pub struct Nim {
    pub combinator: Combinator,
    pub mem: SpragueGrundyMemory,
    pub configuration: NimConfiguration
}

impl Nim
{
    pub fn new(configuration: NimConfiguration) -> Nim
    {
        Nim
        {
            combinator: Combinator::new(),
            mem: SpragueGrundyMemory::new(),
            configuration
        }
    }

    // decides if all nonzero stacks are equal (used in conjunction with
        ↪ is_circuit)
    pub fn stacks_equal(s: &NimState) -> bool
    {
        let mut first_nonzero_value_opt = None;
        for value in s.stacks.iter()
        {
            if *value > 0
            {
                if let Some(first_nonzero_value) = first_nonzero_value_opt
```

```
            {
                if value != first_nonzero_value
                {
                    return false;
                }
            }
            else
            {
                first_nonzero_value_opt = Some(value);
            }
        }
    }
    true
}

// detect if the remaining complex is a circuit
pub fn is_circuit(&mut self, s: &NimState) -> bool
{
    let mut nonzero = Vec::<usize>::with_capacity(self.configuration.
        ↪ num_vertices);
    for (index, value) in s.stacks.iter().enumerate()
    {
        if *value > 0
        {
            nonzero.push(index);
        }
    }

    let n = nonzero.len();
    // must be non-face
    if self.configuration.faces.contains(&nonzero)
    {
        return false;
    }
    for comb in self.combinator.combinations(n, n - 1)
    {
        // should be able to do this because by definition of simplicial
            ↪  complex,
        // if a subset is in the complex, all subsets of that subset are
            ↪  also

        let image = crate::helpers::select(&nonzero, &comb);

        if !self.configuration.faces.contains(&image)
        {
            // all subsets must be contained
            return false;
```

```
        }
    }
    true
}
pub fn minimum_exclusion(set: &HashSet<u32>) -> u32
{
    let mut i = 0;
    return loop{
        if !set.contains(&i)
        {
            break i;
        }
        i += 1;
    }
}

// memoization step, also memoizes symmetrical positions
fn insert_into_sprague_grundy_memory(&mut self, s: &NimState, sg_value:
    ↪ u32)
{
    if self.mem.lookup_table.contains_key(&s.stacks)
    {
        // if one item from the orbit is present, all of them should be.
        return;
    }

    let symmetrical_positions = self.configuration.get_orbit(s.clone());
    let mut min = None;
    let mut argmin = None;

    for (idx, position) in symmetrical_positions.iter().enumerate()
    {
        match min
        {
            None => {
                min = Some(&position.stacks);
                argmin = Some(idx)
            },
            Some(existing) => {
                if position.stacks < *existing
                {
                    min = Some(&position.stacks);
                    argmin = Some(idx);
                }
            }
        }
    }
```

```
    let argmin = argmin.unwrap();

    for (idx, state) in symmetrical_positions.into_iter().enumerate()
    {
        let memory_item = {
            if idx == argmin
            {
                SpragueGrundyMemoryItem
                {
                    sg_value,
                    original: true
                }
            }
            else
            {
                SpragueGrundyMemoryItem
                {
                    sg_value,
                    original: false
                }

            }
        };

        self.mem.lookup_table.entry(state.stacks).or_insert(memory_item)
            ↪ ;
    }
}

pub fn sprague_grundy_value(&mut self, s: &NimState ) -> u32
{
    // base case
    let result: u32 =
        if self.mem.lookup_table.contains_key(&s.stacks)
        {
            self.mem.lookup_table.get(&s.stacks).unwrap().sg_value
        }
        else if self.is_base_nim(&s)
        {
            let mut nim_sum = 0;
            for stack in &s.stacks
            {
                nim_sum ^= *stack;
            }
            nim_sum
        }
        // n * e(C)
```

```
        else if Nim::stacks_equal(&s) && self.is_circuit(&s)
        {
            0
        }
        // recursive step
        else{
            let mut set = HashSet::<u32>::new();
            for mv in self.get_moves(&s).into_iter()
            {
                let new_state = &self.perform_move(s, &mv);
                set.insert(self.sprague_grundy_value( &new_state ));
            }

            Nim::minimum_exclusion(&set)

        };

    self.insert_into_sprague_grundy_memory(s, result);

    result
}

// detects if complex is disjoint, at which point it is equivalent to
    ↪ the original game of nim
pub fn is_base_nim(&self, s: &NimState) -> bool
{
    // if each face has at most one non-empty stack, then nim is not
        ↪ simplicial
    for face in &self.configuration.faces
    {
        let mut num_nonempty: u32 = 0;

        for id in face
        {
            if s.stacks[*id] > 0
            {
                num_nonempty += 1;
            }
            if(num_nonempty) > 1
            {
                return false;
            }
        }

    }
    return true;
}
```

```
// for a given face (combination of vertices), generate possible moves
    ↪ on that face.
fn get_moves_for_combination(s: &NimState, comb: &Vec<usize>) -> Vec<
    ↪ Move>
{
    let mut finished_moves = Vec::<Move>::new();
    let mut unfinished_move_stack = vec![
        (
            0,
            Move {
                removals: vec![]
            }
        )
    ];
    while !unfinished_move_stack.is_empty()
    {
        let current_move = unfinished_move_stack.pop().unwrap();
        let index = current_move.0;
        if index == comb.len()
        {
            // finished, push.
            if !current_move.1.removals.is_empty(){
                finished_moves.push(current_move.1);
            }
            continue;
        }

        // otherwise, push samples for len + 1

        let face_id = comb[index];
        let face_num = s.stacks[face_id];

        // ranges don't go backwards, so this should work fine if
            ↪ face_num is 0
        for num in 1..=face_num {

            let mut new_move = current_move.clone();
            new_move.1.removals.push(Removal{ id: face_id.clone(),
                ↪ num_taken: num });
            new_move.0 += 1; // increase exploration index

            unfinished_move_stack.push(new_move);
        }

    }
    return finished_moves;
```

```
}

// given a state and a symmetry, find the symmetrical state.
pub fn apply_symmetry_to_stacks(s: &NimState, symmetry: &Vec<usize>) ->
    ↪  Vec<u32>
{
    crate::helpers::select(&s.stacks, symmetry)
}

// find the optimal move in a state, harnessing the Sprague-Grundy
    ↪ function
pub fn analytical_move(&mut self, s: &NimState) -> Move
{
    let stall = self.sprague_grundy_value(&s) == 0;
    let mut best_move = None;
    let mut best_sum = None;
    let mut zero_position_found = false;

    for mv in self.get_moves(&s)
    {
        let current_sum: u32 = mv.removals.iter().map( |r| { r.num_taken
            ↪  }).sum();
        if stall
        {
            if best_sum == None || current_sum < best_sum.unwrap()
            {
                best_sum = Some(current_sum);
                best_move = Some(mv);
            }
        }
        else
        {
            let next_state = self.perform_move(&s, &mv);
            let next_sg = self.sprague_grundy_value(&next_state);
            if
                best_sum == None ||
                next_sg == 0 && !zero_position_found ||
                next_sg == 0 && zero_position_found && current_sum >
                    ↪ best_sum.unwrap()
            {
                best_sum = Some(current_sum);
                best_move = Some(mv);
                if next_sg == 0
                {
                    zero_position_found = true;
                }
            }
```

```
            }
        }

        best_move.unwrap()
    }

}

impl game::Game<NimState, Move> for Nim
{
    fn result(&mut self, s: &NimState) -> Option<f64>
    {
        if self.get_moves(&s).is_empty()
        {
            return Some(1.0);
        }
        else
        {
            return None;
        }
    }

    fn get_moves(&mut self, s: &NimState) -> Vec<Move>
    {
        let mut out_set = Vec::<Move>::new();

        for comb in &self.configuration.faces
        {
            out_set.append(&mut Nim::get_moves_for_combination(s, comb));
        }

        out_set.sort();
        out_set.dedup();

        return out_set;
    }

    fn perform_move(&mut self, s: &NimState, m: &Move) -> NimState
    {
        let mut out = s.clone();
        for removal in &m.removals
        {
            let stack_ref = &mut out.stacks[removal.id];

            assert!(
                *stack_ref >= removal.num_taken,
```

```
                "Move should not attempt to remove more sticks ({}) than
                    ↪ there are in the stack ({}).",
                removal.num_taken,
                *stack_ref
            );

            *stack_ref -= removal.num_taken;
        }

        return out;
    }


}


// a single item in the memoization table
#[derive(Debug)]
pub struct Sprague Grundy MemoryItem
{
    pub sg_value: u32,
    pub original: bool // for filtering out symmetries
}

// the memoization table
pub struct Sprague Grundy Memory
{
    pub lookup_table: HashMap<Vec<u32>, Sprague Grundy MemoryItem>
}

impl std::fmt::Display for Sprague Grundy Memory
{
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
    {
        for (key, value) in self.lookup_table.iter()
        {
            write!(f, "{:?}:\t{}\n", key, value.sg_value).unwrap();
        }
        Ok(())
    }
}

impl Sprague Grundy Memory
{
    pub fn new() -> Sprague Grundy Memory
    {
        Sprague Grundy Memory
```

```
        {
            lookup_table: HashMap::<Vec<u32>, SpragueGrundyMemoryItem>::new
                ↪ ()
        }
    }
}

// a game state in nim
#[derive(Clone, Debug, Hash, Eq, PartialEq)]
pub struct NimState
{
    pub stacks: Vec<u32>
}

// a configuration for a nim game (information valid across one game, which
    ↪  contains graph metadata)
#[derive(Debug)]
pub struct NimConfiguration
{
    pub num_vertices: usize,
    pub faces: HashSet<Vec<usize>>,
    pub symmetries: Vec<Vec<usize>>
    /*
    pub generator: Vec<Vec<usize>>
    */
}



impl NimConfiguration
{
    // get all symmetrical positions of a state
    fn get_orbit(&self, s: NimState) -> Vec<NimState>
    {
        // + 1 because identity permutation is intentionally excluded
        let mut orbit = Vec::<NimState>::with_capacity(self.symmetries.len()
            ↪  + 1);
        orbit.push(s);
        for symmetry in &self.symmetries
        {
            let new_state =
                NimState
                {
                    stacks: crate::helpers::select(&orbit[0].stacks, symmetry
                        ↪ )
                };
            orbit.push(new_state);
```

```
    }
    orbit
}

fn is_symmetry(&self, permutation: &Vec<usize>, combinator: &mut
    ↪ Combinator) -> bool
{
    let mut power_set = combinator.all_combinations(permutation.len());
    power_set.retain( |x| !x.is_empty() );
    let power_set = power_set;

    for face in power_set.iter()
    {
        let mut image_of_face = crate::helpers::select(permutation, &
            ↪ face);
        image_of_face.sort();


        let is_face_of_self = self.faces.contains(face);
        let image_is_face_of_self = self.faces.contains(&image_of_face);

        // if face is a face of self and its image is not a face
        // or if face is not a face of self and its image is a face
        if is_face_of_self ^ image_is_face_of_self
        {
            return false;
        }
    }

    true
}

/// Generates all non-identity permutations ‘pi‘ such that permuting ‘
    ↪ faces‘ by ‘pi‘ results in the same
/// simplicial complex structure.
pub fn generate_symmetries(&self) -> Vec<Vec<usize>>
{
    let mut combinator = crate::combinator::Combinator::new();
    let mut permutations = crate::helpers::generate_permutations(self.
        ↪ num_vertices);


    permutations.retain( |x|
        self.is_symmetry(x, &mut combinator)
    );

    // // remove identity permutation
```

```
        permutations.retain( |x|
            *x != (0..(self.num_vertices)).collect::<Vec<usize>>()
        );

        permutations
    }

    /// Generates all non-identity permutations ‘pi‘ such that permuting ‘
        ↪ faces‘ by ‘pi‘ results in the same
    /// simplicial complex structure. These are stored in the ‘symmetries‘
        ↪ field of the current ‘NimConfiguration‘.
    pub fn generate_symmetries_for_self(&mut self)
    {
        self.symmetries = self.generate_symmetries();
    }


}

impl std::fmt::Display for NimState
{
   fn fmt(&self, f: &mut std::fmt::Formatter<’_>) -> std::fmt::Result
   {
       intersperse_for!(
           for stack in self.stacks.iter() => {
               write!(f, "{}", stack).unwrap();
           },
           intersperse => {
               write!(f, "\t").unwrap();
           }
       );

       Ok(())
   }
}
impl game::State for NimState { }



#[derive(Clone, Debug, Hash, Eq, PartialEq, Ord, PartialOrd)]
pub struct Move
{
    pub removals: Vec<Removal>
}

impl std::fmt::Display for Move
{
```

```
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
    {
        for removal in self.removals.iter()
        {
            write!(f, "\t{}\n", removal).unwrap();
        }
        Ok(())
    }
}


impl game::Move for Move {}

#[derive(Clone, Debug, Hash, Eq, PartialEq, Ord, PartialOrd)]
pub struct Removal
{
    pub id: usize,
    pub num_taken: u32
}

impl std::fmt::Display for Removal
{
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
    {
        write!(f, "take {} from stack {}", self.num_taken, self.id)
    }
}

impl std::str::FromStr for Removal
{
    type Err = ();
    fn from_str(s: &str) -> Result<Self, Self::Err>
    {
        let mut tokens = s.split_ascii_whitespace();

        if tokens.next().ok_or(())? != "take"
        {
            return Err(());
        }
        let num_taken_str = tokens.next().ok_or(())?;
        let num_taken = num_taken_str.parse::<u32>().map_err( |_| { })?;
        if tokens.next().ok_or(())? != "from"
        {
            return Err(());
        }
        if tokens.next().ok_or(())? != "stack"
        {
```

```
            return Err(());
        }
        let id_str = tokens.next().ok_or(())?;
        let id = id_str.parse::<usize>().map_err( |_| { } )?;

        Ok( Removal{ id, num_taken } )
    }


}
```

From src/interesting_complexes.rs

```
/// This module contains definitions of many complexes which are
    ↪ interesting to study,
/// many with precalculated symmetries.

use crate::nim::*;
use std::collections::HashSet;
use crate::combinator::Combinator;


pub fn base_nim_3(a: u32, b: u32, c: u32) -> (Nim, NimState)
{
    let stacks = vec![a, b, c];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
    faces.insert(vec![2]);
    (
        Nim::new(NimConfiguration
        {
            num_vertices: 3,
            faces,
            symmetries: vec![vec![2, 1, 0]]
        }),
        NimState {
            stacks
        }
    )

}

pub fn circular_nim(n: usize, k: usize, initial_state: Vec<u32>) -> (Nim,
    ↪ NimState)
{
    assert!(n >= k && k > 0);
    assert_eq!(initial_state.len(), n);
```

```
    let stacks = initial_state;
    let mut combinator = Combinator::new();
    let mut faces = HashSet::<Vec<usize>>::new();

    for starting_index in 0..n
    {
        let k_length_face = {
            let mut temp: Vec<usize> = (starting_index .. starting_index + k
                ↪ )
                .map( |x| x % n )
                .collect();

            temp.sort();
            temp
        };

        for combination in combinator.all_combinations(k)
        {
            faces.insert(crate::helpers::select(&k_length_face, &combination
                ↪ ));
        }
    }

    let mut game = Nim::new(NimConfiguration {
        num_vertices: n,
        faces,
        symmetries: vec![]
    });

    game.configuration.generate_symmetries_for_self();

    (
        game,
        NimState {
            stacks
        }
    )

}

pub fn shriek(a: u32, b: u32, c:u32) -> (Nim, NimState)
{

    let stacks = vec![a, b, c];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
```

```
    faces.insert(vec![2]);
    faces.insert(vec![0, 1]);

    (
        Nim::new(NimConfiguration
        {
            num_vertices: 3,
            faces,
            symmetries: vec![vec![1, 0, 2]]
        }),
        NimState {
            stacks
        }
    )
}

pub fn path_3(a: u32, b: u32, c:u32) -> (Nim, NimState)
{

    let stacks = vec![a, b, c];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
    faces.insert(vec![2]);
    faces.insert(vec![0, 1]);
    faces.insert(vec![1, 2]);
    (
        Nim::new(NimConfiguration
        {
            num_vertices: 3,
            faces,
            symmetries: vec![vec![2, 1, 0]]
        }),
        NimState {
            stacks
        }
    )
}


pub fn joined_path_3_graphs(number_of_paths: usize, starting_value: u32) ->
    ↪  (Nim, NimState)
{
    let num_vertices = number_of_paths * 3;
    // first 3 are the path_3. after that are the dots
    let stacks = vec![starting_value; num_vertices];
```

```
    let mut faces = HashSet::<Vec<usize>>::new();
    for vertex in 0..num_vertices
    {
        // insert all vertices
        faces.insert(vec![vertex]);
    }

    for path_index in 0..number_of_paths
    {
        let offset = path_index * 3;
        faces.insert(vec![offset + 0, offset + 1]);
        faces.insert(vec![offset + 1, offset + 2]);
    }

    let mut game = Nim::new(NimConfiguration {
        num_vertices,
        faces,
        symmetries: vec![]
    });

    game.configuration.generate_symmetries_for_self();
    (
        game,
        NimState {
            stacks
        }
    )
}

pub fn path_3_and_dots(num_dots: usize, starting_value: u32) -> (Nim,
    ↪ NimState)
{
    let num_vertices = num_dots + 3;

    // first 3 are the path_3. after that are the dots
    let stacks = vec![starting_value; num_vertices];

    let mut faces = HashSet::<Vec<usize>>::new();
    for vertex in 0..num_vertices
    {
        // insert all vertices
        faces.insert(vec![vertex]);
    }

    faces.insert(vec![0, 1]);
    faces.insert(vec![1, 2]);
```

```
    let mut game = Nim::new(NimConfiguration {
        num_vertices,
        faces,
        symmetries: vec![]
    });

    game.configuration.generate_symmetries_for_self();

    (
        game,
        NimState {
            stacks
        }
    )
}

pub fn star_graph(num_leaves: usize, starting_value: u32) -> (Nim, NimState
    ↪ )
{
    let num_vertices = num_leaves + 1;
    // stack 0 is the center, because why not.
    let stacks = vec![starting_value; num_vertices];

    let mut faces = HashSet::<Vec<usize>>::new();
    for vertex in 0..num_vertices
    {
        // insert all vertices
        faces.insert(vec![vertex]);

        // insert each spindle of the star
        if vertex > 0
        {
            faces.insert(vec![0, vertex]);
        }
    }

    let mut game = Nim::new(NimConfiguration {
        num_vertices,
        faces,
        symmetries: vec![]
    });

    game.configuration.generate_symmetries_for_self();

    (
        game,
        NimState {
```

```
            stacks
        }
    )
}

pub fn y_graph(a: u32, b: u32, c: u32, d: u32, e: u32) -> (Nim, NimState)
{
    let stacks = vec![a, b, c, d, e];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
    faces.insert(vec![2]);
    faces.insert(vec![3]);
    faces.insert(vec![4]);
    faces.insert(vec![0, 1]);
    faces.insert(vec![1, 2]);
    faces.insert(vec![2, 3]);
    faces.insert(vec![2, 4]);
    (
        Nim::new(NimConfiguration {
            num_vertices: 5,
            faces,
            symmetries: vec![]
        }),
        NimState{
            stacks
        }
    )
}

pub fn square(a: u32, b: u32, c: u32, d: u32) -> (Nim, NimState)
{
    let stacks = vec![a, b, c, d];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
    faces.insert(vec![2]);
    faces.insert(vec![3]);
    faces.insert(vec![0, 1]);
    faces.insert(vec![1, 2]);
    faces.insert(vec![2, 3]);
    faces.insert(vec![0, 3]);
    (
        Nim::new(NimConfiguration {
            num_vertices: 4,
            faces,
            symmetries: vec![]
```

```
        }),
        NimState {
            stacks
        }

    )
}


// a = head; b, c = spine; d, e = front legs; f, g = back legs
pub fn quadruped(a: u32, b: u32, c: u32, d: u32, e: u32, f: u32, g: u32) ->
    ↪  (Nim, NimState)
{
    let stacks = vec![a, b, c, d, e, f, g];
    let mut faces = HashSet::<Vec<usize>>::new();
    faces.insert(vec![0]);
    faces.insert(vec![1]);
    faces.insert(vec![2]);
    faces.insert(vec![3]);
    faces.insert(vec![4]);
    faces.insert(vec![5]);
    faces.insert(vec![6]);
    faces.insert(vec![0, 1]);
    faces.insert(vec![1, 2]);
    faces.insert(vec![1, 3]);
    faces.insert(vec![1, 4]);
    faces.insert(vec![2, 5]);
    faces.insert(vec![2, 6]);

    (
        Nim::new(NimConfiguration {
            num_vertices: 7,
            faces,
            symmetries: vec![]
        }),
        NimState {
            stacks
        }

    )
}


pub fn projective_plane(a: u32, b: u32, c: u32, d: u32, e: u32, f: u32) ->
    ↪ (Nim, NimState)
{
    let mut combinator = Combinator::new();
```

```
let stacks = vec![a, b, c, d, e, f];
let mut faces = HashSet::<Vec<usize>>::new();

// all points
for i in 0..stacks.len()
{
    faces.insert(vec![i]);
}

// all lines
let all_lines = combinator.combinations(6, 2);
for line in all_lines.into_iter()
{
    faces.insert(line);
}

// certain faces
faces.insert(vec![0, 1, 2]);
faces.insert(vec![0, 1, 3]);
faces.insert(vec![0, 2, 4]);
faces.insert(vec![0, 3, 5]);
faces.insert(vec![0, 4, 5]);
faces.insert(vec![1, 2, 5]);
faces.insert(vec![1, 3, 4]);
faces.insert(vec![1, 4, 5]);
faces.insert(vec![2, 3, 4]);
faces.insert(vec![2, 3, 5]);

let symmetries = vec![
    vec![4, 2, 3, 0, 1, 5],
    vec![5, 4, 0, 1, 3, 2],
    vec![5, 2, 3, 1, 0, 4],
    vec![4, 5, 0, 1, 2, 3],
    vec![2, 0, 4, 1, 3, 5],
    vec![2, 4, 3, 0, 5, 1],
    vec![4, 1, 5, 3, 0, 2],
    vec![3, 1, 4, 0, 2, 5],
    vec![3, 2, 5, 4, 0, 1],
    vec![1, 2, 5, 0, 4, 3],
    vec![5, 3, 0, 2, 4, 1],
    vec![2, 1, 0, 5, 4, 3],
    vec![3, 0, 5, 1, 2, 4],
    vec![3, 0, 1, 5, 4, 2],
    vec![3, 4, 2, 1, 5, 0],
    vec![1, 3, 4, 0, 5, 2],
    vec![2, 3, 4, 5, 0, 1],
    vec![2, 1, 5, 0, 3, 4],
```

```
        vec![0, 4, 5, 2, 3, 1],
        vec![0, 5, 3, 4, 1, 2],
        vec![5, 4, 1, 0, 2, 3],
        vec![1, 4, 5, 3, 2, 0],
        vec![5, 1, 4, 2, 0, 3],
        vec![0, 1, 3, 2, 5, 4],
        vec![1, 4, 3, 5, 0, 2],
        vec![4, 0, 5, 2, 1, 3],
        vec![1, 0, 2, 3, 5, 4],
        vec![1, 0, 3, 2, 4, 5],
        vec![3, 4, 1, 2, 0, 5],
        vec![5, 1, 2, 4, 3, 0],
        vec![2, 4, 0, 3, 1, 5],
        vec![3, 5, 2, 0, 4, 1],
        vec![5, 0, 4, 3, 1, 2],
        vec![3, 1, 0, 4, 5, 2],
        vec![4, 2, 0, 3, 5, 1],
        vec![2, 5, 1, 3, 0, 4],
        vec![5, 3, 2, 0, 1, 4],
        vec![4, 1, 3, 5, 2, 0],
        vec![4, 3, 1, 2, 5, 0],
        vec![3, 2, 4, 5, 1, 0],
        vec![0, 5, 4, 3, 2, 1],
        vec![0, 2, 1, 4, 3, 5],
        vec![1, 3, 0, 4, 2, 5],
        vec![5, 2, 1, 3, 4, 0],
        vec![0, 2, 4, 1, 5, 3],
        vec![3, 5, 0, 2, 1, 4],
        vec![4, 3, 2, 1, 0, 5],
        vec![1, 5, 4, 2, 3, 0],
        vec![4, 5, 1, 0, 3, 2],
        vec![1, 5, 2, 4, 0, 3],
        vec![0, 3, 1, 5, 2, 4],
        vec![2, 5, 3, 1, 4, 0],
        vec![4, 0, 2, 5, 3, 1],
        vec![2, 0, 1, 4, 5, 3],
        vec![5, 0, 3, 4, 2, 1],
        vec![0, 3, 5, 1, 4, 2],
        vec![2, 3, 5, 4, 1, 0],
        vec![1, 2, 0, 5, 3, 4],
        vec![0, 4, 2, 5, 1, 3]
    ];

    (
        Nim::new(NimConfiguration {
            num_vertices: 6,
            faces,
```

```
            symmetries
        }),
        NimState {
            stacks
        }
    )

}
```

From src/helpers.rs

```
extern crate rand;
use rand::distributions::Distribution;

/// Randomly select an item from `vector`. The item is removed from `vector
    ↪ `.
/// Returns `Some(the_selected_item)` if present or `None` if the vector is
    ↪  empty.
/// ## Examples
/// ```rust
/// use simplicial_nim::helpers::sample_and_pop;
///
/// let mut x = Vec::<u32>::new();
/// let mut y: Vec<u32> = vec![3];
///
/// let mut rng = rand::thread_rng();
/// assert_eq!(sample_and_pop(&mut rng, &mut x), None);
/// assert_eq!(sample_and_pop(&mut rng, &mut y), Some(3));
/// assert_eq!(sample_and_pop(&mut rng, &mut y), None);
/// ```
pub fn sample_and_pop<T>(rng: &mut rand::rngs::ThreadRng, vector: &mut Vec<
    ↪ T>) -> Option<T>
{
    if vector.is_empty()
    {
        return None
    }
    Some(vector.swap_remove(
        rand::distributions::Uniform::from(0..vector.len()).sample(rng)
    ))
}


pub fn sample_and_consume<T>(rng: &mut rand::rngs::ThreadRng, mut vector:
    ↪ Vec<T>)-> Option<T>
{
    if vector.is_empty()
    {
        return None
```

```
    }
    Some(vector.swap_remove(
        rand::distributions::Uniform::from(0..vector.len()).sample(rng)
    ))
}

pub fn select<T>(source_vector: &Vec<T>, indices: &Vec<usize>) -> Vec<T>
    where T: Clone
{
    let mut out = Vec::<T>::with_capacity(indices.len());

    for index in indices.iter()
    {
        out.push(source_vector[*index].clone());
    }

    out
}

pub fn factorial_upper_bound(n: usize) -> usize
{
    let n = n as f64;
    use std::f64::consts::E as e;
    use std::f64::consts::PI as pi;

    // Ramanujan order 1 / n^5 approximation
    let approx = (2.0 * pi * n).sqrt() * (n / e).powf(n) * e.powf( (1.0 /
        ↪ (12.0 * n)) * (1.0 - 1.0 / (30.0 * n * n) ) );

    return approx.ceil() as usize
}

pub fn generate_permutations(n: usize) -> Vec<Vec<usize>>
{
    let mut output = Vec::<Vec<usize>>::with_capacity(factorial_upper_bound
        ↪ (n));
    let mut identity = (0..n).collect();
    inner_generate_permutations(n, &mut identity, &mut output);

    output
}

fn inner_generate_permutations(k: usize, a: &mut Vec<usize>, output: &mut
    ↪ Vec<Vec<usize>>)
{
    if k == 1
    {
```

```
        output.push(a.clone());
        return;
    }

    inner_generate_permutations(k - 1, a, output);

    for i in 0..(k - 1)
    {
        if k % 2 == 0
        {
            a.swap(i, k - 1);
        }
        else
        {
            a.swap(0, k - 1);
        }

        inner_generate_permutations(k - 1, a, output);
    }

}

pub fn overwrite_file(name: String) -> std::io::BufWriter<std::fs::File>
{
    let opt = std::fs::OpenOptions::new()
    .write(true)
    .truncate(true)
    .create(true)
    .open(name)
    .unwrap();

    std::io::BufWriter::new(opt)
}


fn read_sg_data(path: String) -> Vec<(Vec<u32>, u32)>
{
    let mut out = Vec::<(Vec<u32>, u32)>::new();

    let sg_string = std::fs::read_to_string(
        path
    ).unwrap();

    for line in sg_string.lines()
    {
        let mut position = Vec::<u32>::new();
        for index in line.split_ascii_whitespace()
```

```
        {
            position.push(index.parse::<u32>().unwrap());
        }
        let sg_value = position.pop().unwrap();
        out.push( (position, sg_value) );
    }

    out

}

pub fn lattice(dimension: usize, depth: u32) -> Vec<Vec<u32>>
{
    let mut out = Vec::<Vec<u32>>::with_capacity((depth as usize).pow(
        ↪ dimension as u32));
    inner_lattice(dimension, depth, Vec::<u32>::with_capacity(dimension), &
        ↪ mut out);
    out
}

fn inner_lattice(dimension: usize, depth: u32, working_vector: Vec<u32>,
    ↪ out_set: &mut Vec<Vec<u32>>)
{
    if working_vector.len() == dimension
    {
        out_set.push(working_vector);
        return;
    }

    for i in 0..depth
    {
        let mut child_vector = working_vector.clone();
        child_vector.push(i);
        inner_lattice(dimension, depth, child_vector, out_set);
    }
}

pub fn read_symmetries_removed(name: &str) -> Vec<(Vec<u32>, u32)>
{
    read_sg_data(
        format!("results/data/{}/symmetries_removed/sprague_grundy", name)
    )
}

pub fn read_sprague_grundy(name: &str) -> Vec<(Vec<u32>, u32)>
{
    read_sg_data(
```

```
        format!("results/data/{}/all/sprague_grundy", name)
    )
}

/// `ListExplorer` is an iterator that infinitely explores lists of a
    ↪ finite number `k` of options.
/// It does this by exploring the list corresponding to numbers 1, 2, ...
    ↪ in base `k`.
pub struct ListExplorer<T>
    where T: Clone
{
    num_options: usize,
    options_list: Vec<T>,
    current_list: Vec<usize>,
    step: usize
}

impl<T> ListExplorer<T>
    where T: Clone
{
    pub fn new(options_list: Vec<T>, current_list: Vec<usize>, step: Option
        ↪ <usize>) -> ListExplorer<T>
    {
        let step = step.unwrap_or(1);
        assert!(step > 0, "Cannot have step of zero in list explorer.");
        ListExplorer
        {
            num_options: options_list.len(),
            options_list,
            current_list,
            step
        }
    }

    fn increment(&mut self, index: usize, amount: usize) -> usize
    {
        if index == self.current_list.len()
        {
            self.current_list.push(amount - 1);
        }
        else
        {
            self.current_list[index] += amount;
        }

        let multiple = self.current_list[index] / self.num_options;
        let modulus = self.current_list[index] - self.num_options * multiple
```

```
            ↪ ;

        self.current_list[index] = modulus;

        multiple
    }

}

impl<T> Iterator for ListExplorer<T>
    where T: Clone
{
    type Item = Vec<T>;
    fn next(&mut self) -> Option<Self::Item>
    {
        let out = select(&self.options_list, &self.current_list);

        let mut carry = self.increment(0, self.step);

        let mut index_to_check = 0;

        while
            index_to_check < self.current_list.len() &&
            carry != 0
        {
            carry = self.increment(index_to_check + 1, carry);
            index_to_check += 1;
        }

        Some(out)
    }
}

#[macro_export]
macro_rules! intersperse_while {
    ( while $condition:expr => { $($while_stmt:stmt;)* }, intersperse => {
        ↪ $($intersperse_stmt:stmt;)* } ) => {
        {
            let mut first = true;
            while $condition {
                if first
                {
                    first = false;
                }
                else
                {
                    $($intersperse_stmt)*
```

```
            }
            $($while_stmt)*
        }
    }
};
}

#[macro_export]
macro_rules! intersperse_for {
    ( for $pattern:pat in $iterator:expr => { $($for_stmt:stmt;)* },
        ↪ intersperse => { $($intersperse_stmt:stmt;)* } ) => {
        {
            let mut first = true;
            for $pattern in $iterator {
                if first
                {
                    first = false;
                }
                else
                {
                    $($intersperse_stmt)*
                }
                $($for_stmt)*
            }
        }
    };
}

#[cfg(test)]
mod tests{
    use super::*;
    #[test]
    fn test_list_explorer()
    {
        let mut count = 0;
        for i in ListExplorer::new( (0..16).collect::<Vec<u32>>(), vec![1],
            ↪ Some(3))
        {
            if count == 32
            {
                break;
            }
            println!("{:?}", i);
            count += 1;

        }
    }
```

```
    #[test]
    fn test_intersperse_while() {

        let mut i = 0;
        intersperse_while! (
            while i < 10 => {
                println!("{}", i);
                i += 1;
            },
            intersperse => {
                println!("moo");
            }
        )

    }

    #[test]
    fn test_intersperse_for() {


        intersperse_for! (
            for i in 0..10 => {
                print!("{}", i);


            },
            intersperse => {
                print!(",");
            }
        )

    }

}
```

From src/function_suggester.rs

```
extern crate rand;

use std::collections::{VecDeque};
use rand::thread_rng;

#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Op
{
    Plus,
    Minus,
    Mul,
    Div,
```

```
    And,
    Or,
    Xor,
    Not,
    Zero,
    One,
    LeftShift,
    RightShift,
    BooleanEq,
    BooleanNot,
    Push(usize) // Load the number of chips in stack #x.
}

fn all_ops(num_registers: usize) -> Vec<Op>
{
    let mut all_ops = vec![
        Op::Plus,
        Op::Minus,
        Op::Mul,
        Op::Div,
        Op::And,
        Op::Or,
        Op::Xor,
        Op::Not,
        Op::Zero,
        Op::One,
        Op::BooleanEq,
        Op::BooleanNot,
        Op::LeftShift,
        Op::RightShift
    ];

    for i in 0..num_registers
    {
        all_ops.push(Op::Push(i));
    }

    all_ops
}

fn take_action(registers: &Vec<u32>, stack: &mut Vec<u32>, op: &Op) ->
    ↪ Result<(), ()>
{
    match op
    {
        Op::Plus => {
            let op1 = stack.pop().ok_or(())?;
```

```
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 + op1);
    Ok(())
},
Op::Minus => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 - op1);
    Ok(())
},
Op::Mul => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 * op1);
    Ok(())
},
Op::Div => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    if op1 == 0
    {
        Err(())
    }
    else
    {
        stack.push(op0 / op1);
        Ok(())
    }

},
Op::And => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 & op1);
    Ok(())

},
Op::Or => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 | op1);
```

```
    Ok(())

},
Op::Xor => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    stack.push(op0 ^ op1);
    Ok(())
},
Op::Not => {
    let op0 = stack.pop().ok_or(())?;

    stack.push( !op0 );
    Ok(())

},
Op::Zero => {
    stack.push(0);
    Ok(())
},
Op::One => {
    stack.push(1);
    Ok(())
},
Op::BooleanEq => {
    let op1 = stack.pop().ok_or(())?;
    let op0 = stack.pop().ok_or(())?;

    let result = if op0 == op1{
        1
    } else
    {
        0
    };
    stack.push(result);
    Ok(())
},
Op::BooleanNot => {
    let op0 = stack.pop().ok_or(())?;

    let result = if op0 == 0{
        1
    } else
    {
        0
    };
```

```
            stack.push(result);
            Ok(())

        },
        Op::LeftShift => {
            let op1 = stack.pop().ok_or(())?;
            let op0 = stack.pop().ok_or(())?;

            stack.push(op0 << op1);
            Ok(())

        },
        Op::RightShift => {
            let op1 = stack.pop().ok_or(())?;
            let op0 = stack.pop().ok_or(())?;

            stack.push(op0 >> op1);
            Ok(())
        },
        Op::Push(i) => {
            let value = *registers.get(*i).ok_or(())?;
            stack.push(value);
            Ok(())
        }
    }
}

fn program_is_correct( program: &Vec<Op>, data: &Vec<Vec<u32>>, answers: &
    ↪ Vec<u32> ) -> Result<bool, ()>
{
    for (state, target) in data.iter().zip(answers.iter())
    {
        let mut stack = vec![];

        for op in program.iter()
        {
            take_action(state, &mut stack, &op)?;
        }

        let predicted = stack.pop().ok_or(())?;

        if predicted != *target
        {
            return Ok(false);
        }
    }
```

```
    Ok(true)
}

fn program_is_correct_wrt_symmetries( program: &Vec<Op>, data: &Vec<Vec<u32
    ↪ >>, answers: &Vec<u32>, symmetries: &Vec<Vec<usize>>) -> Result<bool
    ↪ , ()>
{
    // a false positive is immediate disqualification
    // a false negative is okay, as long as the program works on a
        ↪ symmetrical position

    let mut stack = vec![];

    for (state, target) in data.iter().zip(answers.iter())
    {
        // determine result for original
        stack.clear();

        for op in program.iter()
        {
            take_action(state, &mut stack, &op)?;
        }

        let predicted_zero_pos = stack.pop().ok_or(())? == 0;
        let target_is_zero_pos = *target == 0;

        if predicted_zero_pos
        {
            if target_is_zero_pos // true positive
            {
                continue; // this one's okay, but has to work on all
                    ↪ positions
            }
            else // false positive
            {
                return Ok(false);
            }

        }

        // examine symmetrical positions.

        for symmetry in symmetries
        {
            let state = crate::helpers::select(state, symmetry);
            stack.clear();
```

```
            for op in program.iter()
            {
                take_action(&state, &mut stack, &op)?;
            }

            let predicted_zero_pos = stack.pop().ok_or(())? == 0;

            if predicted_zero_pos
            {
                if target_is_zero_pos // true positive
                {
                    continue; // this one's okay, but has to work on all
                        ↪ positions
                }
                else // false positive
                {
                    return Ok(false);
                }

            }
        }

        if target_is_zero_pos // must not have given a positive result for
            ↪ any position in the orbit
        {
            return Ok(false);
        }
        // if the if statement above didn't run, then it should be a true
        // negative result, and the appropriate action would be 'continue'
    }


    Ok(true)
}

pub fn suggest_function_bfs_memory(data: Vec<Vec<u32>>, answers: Vec<u32>,
    ↪ num_threads: Option<usize>) -> Vec<Op>
{
    use std::sync::{mpsc, Arc};
    use std::thread;

    let num_threads = num_threads.unwrap_or(1);
    let ops_list = all_ops(data[0].len());
    let (tx, rx) = mpsc::channel();
    let data_arc = Arc::new(data);
    let answers_arc = Arc::new(answers);
```

```
for i in 0..num_threads
{
    let data = Arc::clone(&data_arc);
    let answers = Arc::clone(&answers_arc);
    let tx = tx.clone();
    let ops_list = ops_list.clone();

    thread::spawn(move|| {
        // List explorer handles the base-num_instructions number
            ↪ exploration order
        // wrapping it up inside a single iterator. It will infinitely
            ↪ explore programs
        // from shorter length to longer length, without consuming much
            ↪ memory.

        // See helpers module for implementation.
        let program_iterator = crate::helpers::ListExplorer::new(
            ↪ ops_list, vec![i], Some(num_threads));
        let mut program_length: usize = 0;
        'search: for program in program_iterator
        {
            if program.len() > program_length
            {
                program_length = program.len();
                println!("Now exploring programs of length {}.",
                    ↪ program_length);
            }

            // if program compiles (is Ok) and is correct, then we've got
                ↪  a solution
            if let Ok(is_correct) = program_is_correct(&program, &data, &
                ↪ answers) // if it doesn't compile now, it won't
                ↪ compile later
            {
                if is_correct
                {
                    let _result = tx.send(program);
                    break 'search;
                }
            }

        }

    });
}

let result = rx.recv().unwrap();
```

```
    println!("!!!\n{:?}", result);
    result
}

pub fn suggest_function_bfs_memory_wrt_symmetries(data: Vec<Vec<u32>>,
    ↪ answers: Vec<u32>, symmetries: Vec<Vec<usize>>, num_threads: Option<
    ↪ usize>) -> Vec<Op>
{
    use std::sync::{mpsc, Arc};
    use std::thread;

    let num_threads = num_threads.unwrap_or(1);
    let ops_list = all_ops(data[0].len());
    let (tx, rx) = mpsc::channel();
    let data_arc = Arc::new(data);
    let answers_arc = Arc::new(answers);
    let symmetries_arc = Arc::new(symmetries);

    for i in 0..num_threads
    {
        let data = Arc::clone(&data_arc);
        let answers = Arc::clone(&answers_arc);
        let symmetries = Arc::clone(&symmetries_arc);
        let tx = tx.clone();
        let ops_list = ops_list.clone();

        thread::spawn(move|| {
            let program_iterator = crate::helpers::ListExplorer::new(
                ↪ ops_list, vec![i], Some(num_threads));
            let mut program_length: usize = 0;
            'search: for program in program_iterator
            {
                if program.len() > program_length
                {
                    program_length = program.len();
                    println!("Now exploring programs of length {}.",
                        ↪ program_length);
                }

                if let Ok(is_correct) = program_is_correct_wrt_symmetries(&
                    ↪ program, &data, &answers, &symmetries) // if it doesn'
                    ↪ t compile now, it won't compile later
                {
                    if is_correct
                    {
                        let _result = tx.send(program);
                        break 'search;
```

```
                }
            }

        }

    });
}

let result = rx.recv().unwrap();
println!("!!!\n{:?}", result);
result
}

pub fn suggest_function_bfs_cpu(data: Vec<Vec<u32>>, answers: Vec<u32>,
    ↪ initial_guess: Option<Vec<Op>>) -> Vec<Op>
{
    let start_state = if let Some(initial_guess) = initial_guess
    {
        let is_correct = program_is_correct(&initial_guess, &data, &answers)
            ↪ .unwrap();
        if is_correct
        {
            println!("!!!\n{:?}", initial_guess);
            return initial_guess;
        }
        initial_guess
    }
    else
    {
        vec![]
    };

    let mut program_length: usize = 0;
    let mut exploration_stack = VecDeque::<Vec<Op>>::new();

    exploration_stack.push_back(
        start_state
    );

    let actions = all_ops(data[0].len());
    let mut rng = thread_rng();

    loop
    {
        let to_explore = exploration_stack.pop_front().unwrap(); // there
            ↪ are infinite possible programs
        if to_explore.len() > program_length
```

```
        {
            program_length = to_explore.len();
            println!("Now exploring programs of length {}.", program_length)
                ↪ ;
        }

        let mut next_actions = actions.clone();
        while !next_actions.is_empty()
        {
            // pick a random next instruction
            let next_action = crate::helpers::sample_and_pop::<Op>(&mut rng,
                ↪  &mut next_actions).unwrap();
            let mut new_program = to_explore.clone();
            new_program.push(next_action);

            // if program does not violate the stack
            if let Ok(is_correct) = program_is_correct(&new_program, &data,
                ↪ &answers) // if it doesn't compile now, it won't compile
                ↪ later
            {
                // and is correct
                if is_correct
                {
                    // then a solution has been found
                    println!("!!!\n{:?}", new_program);
                    return new_program;
                }
                // otherwise, later explore the programs that contain it as a
                    ↪  prefix
                exploration_stack.push_back(new_program);
            }
            // if the program doesn't compile, no programs that contain it
                ↪ as a prefix will be explored.
        }

    }


}


// Tests for the function suggester must all be run in release mode.
#[cfg(test)]
#[cfg(not(debug_assertions))]
mod tests
{
    use super::*;
```

```
#[ignore]
#[test]
fn suggest_function_for_projective_plane()
{
    let sprague_grundy_values = crate::helpers::read_sprague_grundy("
        ↪ projective_plane");
    // let sprague_grundy_values = crate::helpers::
        ↪ read_symmetries_removed("projective_plane"); // for
        ↪ wrt_symmetries version
    let mut data = Vec::<Vec<u32>>::new();
    let mut answers = Vec::<u32>::new();

    for (datum, answer) in sprague_grundy_values
    {
        data.push(datum);
        // answers.push(answer); // sg function
        answers.push(
            if answer == 0
            {
                0
            }
            else
            {
                1
            }
        );
    }

    // Starting guess-- only works for CPU-optimized version:

    // suggest_function_bfs_cpu(data, answers, Some(vec![
    // // first three zero
    // Op::Push(0),
    // Op::Push(1),
    // Op::Push(2),
    // Op::Or,
    // Op::Or,
    // Op::Zero,
    // Op::BooleanEq,

    // // last three equal
    // Op::Push(3),
    // Op::Push(4),
    // Op::Xor,
    // Op::Push(4),
    // Op::Push(5),
```

```
    // Op::Xor,
    // Op::Or,
    // Op::Zero,
    // Op::BooleanEq,

    // // combine
    // Op::And
    // ]));

    suggest_function_bfs_memory(data, answers, Some(4));
    // suggest_function_bfs_cpu(data, answers, None);

    // let (game, _) = crate::interesting_complexes::projective_plane(1,
        ↪  1, 1, 1, 1, 1);
    // suggest_function_bfs_memory_wrt_symmetries(data, answers, game.
        ↪ configuration.symmetries, Some(3));
}

#[ignore]
#[test]
fn suggest_function_for_path_3()
{
    let sprague_grundy_values = crate::helpers::read_sprague_grundy("
        ↪ path_3");
    let mut data = Vec::<Vec<u32>>::new();
    let mut answers = Vec::<u32>::new();

    for (datum, answer) in sprague_grundy_values
    {
        data.push(datum);
        // answers.push(answer); // sg function
        answers.push(
            // zero position function described in the text
            if answer == 0
            {
                0
            }
            else
            {
                1
            }
        );
    }

    // suggest_function_bfs_cpu(data, answers, None);
    suggest_function_bfs_memory(data, answers, Some(4));
    // suggest_function_bfs_memory_wrt_symmetries(data, answers, vec![
```

```
        ↪ vec![2, 1, 0]], Some(4));

}

#[ignore]
#[test]
fn suggest_function_for_shriek()
{
    let sprague_grundy_values = crate::helpers::read_sprague_grundy("
        ↪ shriek");
    // let symmetries_removed = crate::helpers::read_symmetries_removed
        ↪ ("shriek");
    let mut data = Vec::<Vec<u32>>::new();
    let mut answers = Vec::<u32>::new();

    for (datum, answer) in sprague_grundy_values
    {
        data.push(datum);
        answers.push(answer); // sg function
    }

    // suggest_function_bfs_cpu(data, answers, None);
    suggest_function_bfs_memory(data, answers, Some(4));
    // suggest_function_bfs_memory_wrt_symmetries(data, answers, vec![
        ↪ vec![1, 0, 2]], Some(4));

}

#[ignore]
#[test]
fn suggest_function_for_star_graphs()
{

    use std::time::{Duration, Instant};
    use std::io::Write;

    let mut results_file = crate::helpers::overwrite_file(String::from("
        ↪ results/compiled/star_programs"));

    for star_size in 1..3
    {
        writeln!(results_file, "star graph with {} leaves:", star_size).
            ↪ unwrap();

        let sprague_grundy_values = crate::helpers::read_sprague_grundy
            ↪ (&format!("star/{}", star_size)[..]);
        let mut data = Vec::<Vec<u32>>::new();
```

```
        let mut answers = Vec::<u32>::new();

        for (datum, answer) in sprague_grundy_values
        {
            data.push(datum);
            // answers.push(answer); // sg function
            answers.push(
                if answer == 0
                {
                    0
                }
                else
                {
                    1
                }
            );
        }

        // suggest_function_bfs_cpu(data, answers, None);
        let start_time = Instant::now();
        let satisfying_program = suggest_function_bfs_memory(data,
            ↪ answers, Some(4));
        let end_time = Instant::now();
        let elapsed_time: Duration = end_time - start_time;

        writeln!(results_file, "\tfound solution in {:?}", elapsed_time)
            ↪ .unwrap();
        write!(results_file, "\t{:?}\n\n", satisfying_program).unwrap();

        results_file.flush().unwrap(); // inside of the loop because it'
            ↪ s nice to check on its progress
    }

}

#[ignore]
#[test]
fn suggest_function_for_circular_nim()
{

    use std::time::{Duration, Instant};
    use std::io::Write;

    let mut results_file = crate::helpers::overwrite_file(String::from("
        ↪ results/compiled/circular_nim_programs"));

    let nks = vec![
```

```
    (2, 1),
    (2, 2),
    (3, 1),
    (3, 2),
    (3, 3),
    (4, 1),
    (4, 2),
    (4, 3),
    (4, 4)
];


for (n, k) in nks.iter()
{
    writeln!(results_file, "CN({}, {}):", n, k).unwrap();
    println!("exploring CN({}, {})", n, k);

    let symmetries_removed = crate::helpers::read_sprague_grundy(&
        ↪ format!("circular_nim/{}_{}", n, k)[..]);
    let mut data = Vec::<Vec<u32>>::new();
    let mut answers = Vec::<u32>::new();

    for (datum, answer) in symmetries_removed
    {
        data.push(datum);
        // answers.push(answer); // sg function
        answers.push(
            if answer == 0
            {
                0
            }
            else
            {
                1
            }
        );
    }

    // suggest_function_bfs_cpu(data, answers, None);
    let start_time = Instant::now();
    let satisfying_program = suggest_function_bfs_memory(data,
        ↪ answers, Some(4));
    let end_time = Instant::now();
    let elapsed_time: Duration = end_time - start_time;

    writeln!(results_file, "\tfound solution in {:?}", elapsed_time)
        ↪ .unwrap();
```

```
            write!(results_file, "\t{:?}\n\n", satisfying_program).unwrap();

            results_file.flush().unwrap(); // inside of the loop because it'
                ↪ s nice to check on its progress
        }

    }

}
```

From src/combinator.rs

```
use std::collections::HashMap;

/// ‘Combinator‘ structs memoize both all possible combinations of n
    ↪ elements choosing k items, as well
/// as all possible combinations of n elements choosing any number of items
    ↪ .
pub struct Combinator
{
    nk_lookup_table: HashMap< (usize, usize), Vec<Vec<usize>> >,
    all_lookup_table: HashMap< usize, Vec<Vec<usize>> >
}
impl Combinator
{
    pub fn new() -> Combinator
    {
        Combinator
        {
            nk_lookup_table: HashMap::< (usize, usize), Vec<Vec<usize>> >::
                ↪ new(),
            all_lookup_table: HashMap::< usize, Vec<Vec<usize>> >::new()
        }
    }
    pub fn all_combinations(&mut self, n: usize) -> Vec::<Vec<usize>>
    {
        if let Some(combs) = self.all_lookup_table.get(&n)
        {
            return combs.clone();
        }

        let base: usize = 2;

        let mut out_combs = Vec::<Vec<usize>>::with_capacity(base.pow( n as
            ↪ u32 ));
        for k in 0..=n
        {
            for comb in self.combinations(n, k).into_iter()
            {
```

```
                out_combs.push(comb);
            }
        }
        self.all_lookup_table.insert(n, out_combs.clone());
        return out_combs;
    }

    pub fn combinations(&mut self, n: usize, k: usize) -> Vec<Vec<usize>>
    {
        if k > n
        {
            return vec![];
        }
        if let Some(combs) = self.nk_lookup_table.get( &(n, k) )
        {
            return combs.clone();
        }
        if k == 0
        {
            // 1 empty combination
            self.nk_lookup_table.insert( (n, k), vec![ vec![] ] );
            return vec![ vec![] ];
        }
        if k == n
        {
            let out_combs: Vec<Vec<usize>> = vec![ (0..n).collect() ];
            self.nk_lookup_table.insert( (n, k), out_combs.clone());
            return out_combs;
        }

        let mut out_combs = self.combinations(n - 1, k).clone(); // all
            ↪ combinations from the right

        for mut comb in self.combinations(n - 1, k - 1).into_iter()
        {
            comb.push(n - 1); // insert last node
            out_combs.push(comb);
        }

        self.nk_lookup_table.insert( (n, k), out_combs.clone());
        out_combs
    }
}

#[cfg(test)]
mod tests
{
```

```
    use super::*;
    #[test]
    fn test_combinations()
    {
        let mut combinator = Combinator::new();
        let combinations = combinator.combinations(3, 1);
        assert_eq!(combinations, vec![vec![0], vec![1], vec![2]]);

        let combinations = combinator.combinations(3, 2);
        assert_eq!(combinations, vec![
            vec![0, 1],
            vec![0, 2],
            vec![1, 2]
        ]);

        let combinations = combinator.all_combinations(3);
        assert_eq!(combinations, vec![
            vec![],
            vec![0],
            vec![1],
            vec![2],
            vec![0, 1],
            vec![0, 2],
            vec![1, 2],
            vec![0, 1, 2]
        ]);



    }
}
```

From src/ai/mod.rs

```
pub mod agent;
pub mod game;
pub mod random_agent;
pub mod mcts_agent;
pub mod game_tree;
pub mod game_runner;
```

From src/ai/agent.rs

```
use crate::ai::game::*;

pub trait Agent<S: State, M: Move>
{
    fn recommend_move(&mut self, game: &mut Box<dyn Game<S, M>>, state: &S)
        ↪  -> M;
}
```

From src/ai/game.rs

```
// use std::collections::HashSet;
use std::hash::Hash;

pub trait State: Clone + Hash + PartialEq + Eq + std::fmt::Display {}

pub trait Move: Clone + Hash + PartialEq + Eq + std::fmt::Display {}

pub trait Game<S: State, M: Move>
{
    // all available moves from a certain state
    fn get_moves(&mut self, state:&S) -> Vec<M>;

    // state transition function
    fn perform_move(&mut self, state:&S, m: &M) -> S;

    // reward from the game (from the previous player's point of view),
        ↪ None if not finished
    fn result(& mut self, state:&S) -> Option<f64>;
}
```

From src/ai/game_tree.rs

```
use crate::ai::game::{State, Move};

pub trait Info: Clone { }

pub struct GameTreeNode<S: State, M:Move, I: Info>
{
    pub state: S,
    pub previous_move: Option<M>,
    pub extra_information: I,
    pub children: Vec<GameTreeNode<S, M, I>>
}

impl<S:State, M:Move, I:Info> GameTreeNode<S, M, I>
{
    pub fn new(initial_state: S, previous_move: Option<M>,
        ↪ initial_information: I) -> GameTreeNode<S, M, I>
    {
        GameTreeNode {
            state: initial_state,
            previous_move: previous_move,
            extra_information: initial_information,
            children: vec![]
        }
    }
```

```
}
```

From src/ai/mcts_agent.rs

```
extern crate rand;

use std::time::{Duration, Instant};
use rand::seq::SliceRandom;
use rand::distributions::Distribution;
use std::marker::PhantomData;

use crate::ai::agent::Agent as Agent;
use crate::ai::game::State as State;
use crate::ai::game::Move as Move;
use crate::ai::game::Game as Game;
use crate::ai::game_tree::{GameTreeNode, Info};


#[derive(Clone)]
struct MctsInfo<M: Move>
{
    pub reward: f64,
    pub count: u64,
    pub depth: u64,
    pub moves_left: Vec<M>
}

impl<M: Move> Info for MctsInfo<M> {}



pub struct MctsAgent<S: State, M: Move>
{
    pub alternating_reward: bool,
    pub seconds_alotted: u64,
    pub rng: rand::prelude::ThreadRng,
    state_type: PhantomData<S>,
    move_type: PhantomData<M>,
}

impl<S: State, M: Move> MctsAgent<S, M>
{
    pub fn new(alternating_reward: bool, seconds_alotted: u64) -> MctsAgent
        ↪ <S, M>
    {
        MctsAgent
        {
            alternating_reward,
            seconds_alotted,
```

```
        rng: rand::thread_rng(),
        state_type: PhantomData,
        move_type: PhantomData,
    }
}

fn evaluate_terminal(&self, game: &mut Box<dyn Game<S, M>>, state: &S,
    ↪ depth: u64) -> Option<f64>
{
    let raw_result = game.result(&state);
    if let Some(raw_result) = raw_result
    {
        let turn = ((depth - 1) % 2) as f64;
        Some ((2.0 * (1.0 - turn) - 1.0) * raw_result)
    }
    else
    {
        None
    }
}

fn best_child<'a>(&self, node: &'a mut GameTreeNode<S, M, MctsInfo<M>>)
    ↪  -> Option<&'a mut GameTreeNode<S, M, MctsInfo<M>>>
{
    let mut t: f64 = 0.0;
    for child in &node.children { t += child.extra_information.count as
        ↪ f64; }
    let t = t;

    let positive_reward = !self.alternating_reward || node.
        ↪ extra_information.depth % 2 == 0;
    let multiplier =
    if positive_reward {
        1.0
    }
    else {
        -1.0
    };

    let mut best_ucb_value = f64::NEG_INFINITY;
    let mut best_child = None;

    for child in &mut node.children
    {
        let q = child.extra_information.reward;
        let n = child.extra_information.count as f64;
        let ucb_value = multiplier * q / n + f64::sqrt(2.0 * t.ln() / n)
```

```
            ↪ ;
        if ucb_value > best_ucb_value
        {
            best_ucb_value = ucb_value;
            best_child = Some(child);
        }
    }

    return best_child
}
fn update(&mut self, game: &mut Box<dyn Game<S, M>>, node: &mut
    ↪ GameTreeNode<S, M, MctsInfo<M>>) -> (f64, u64)
{
    // base case: at a not fully expanded node
    let num_moves_left = node.extra_information.moves_left.len();
    if num_moves_left > 0
    {
        // pick move to make
        let move_to_explore = node.extra_information.moves_left.
            ↪ swap_remove(
            rand::distributions::Uniform::from(0..num_moves_left).sample
                ↪ (&mut self.rng)
        );
        // add child
        let new_state = game.perform_move(&node.state, &move_to_explore)
            ↪ ;
        let new_moves = game.get_moves(&new_state);
        let child_to_add = GameTreeNode::<S, M, MctsInfo<M>>:: new(
            new_state,
            Some(move_to_explore),
            MctsInfo::<M> {
                reward: 0.0,
                count: 0,
                depth: node.extra_information.depth + 1,
                moves_left: new_moves
            }
        );

        node.children.push(child_to_add);

        // simulate playout
        let last_index:usize = node.children.len() - 1;
        let newly_added_node = &mut node.children[last_index];
        let mut current_state = newly_added_node.state.clone();
        let mut current_depth = newly_added_node.extra_information.depth
            ↪ ;
        while let None = game.result(&current_state)
```

```
            {
                let available_moves = game.get_moves(&current_state);
                let random_move = available_moves.choose(&mut self.rng).
                    ↪ unwrap();
                current_state = game.perform_move(&current_state, random_move
                    ↪ );
                current_depth += 1;
            }

            let simulation_result = (self.evaluate_terminal(game, &
                ↪ current_state, current_depth).unwrap(), 1);
            newly_added_node.extra_information.reward = simulation_result.0;
            newly_added_node.extra_information.count = 1;



            return simulation_result;

        }

        let result =
        if let Some(value) = self.evaluate_terminal(game, &node.state, node.
            ↪ extra_information.depth)
        {
            (value, 1)
        }
        else
        {
            self.update(game, &mut self.best_child(node).unwrap())
        };


        // update
        node.extra_information.reward += result.0;
        node.extra_information.count += result.1;

        return result;

    }


}

impl<S: State, M: Move> Agent<S, M> for MctsAgent<S, M>
{
    fn recommend_move(&mut self, game: &mut Box<dyn Game<S, M>>, state: &S)
        ↪  -> M
```

```
    {
        let mut root = GameTreeNode::<S, M, MctsInfo<M>>::new(state.clone(),
            ↪  None, MctsInfo { reward: 0.0, count: 0, depth: 0, moves_left
            ↪ : game.get_moves(&state) } );

        let start_time = Instant::now();
        let mut current_time = Instant::now();
        let budget = Duration::new(self.seconds_alotted, 0);
        while current_time.duration_since(start_time) < budget
        {
            &self.update(game, &mut root);
            current_time = Instant::now();
        }

        return self.best_child(&mut root).unwrap().previous_move.clone().
            ↪ unwrap();
    }
}
```

From src/ai/random_agent.rs

```
extern crate rand;

use rand::seq::SliceRandom;
use std::marker::PhantomData;
use crate::ai::agent::Agent as Agent;
use crate::ai::game::State as State;
use crate::ai::game::Move as Move;
use crate::ai::game::Game as Game;


// pub struct RandomAgent{}
pub struct RandomAgent<S: State, M: Move>
{
    state_type: PhantomData<S>,
    move_type: PhantomData<M>
}

impl<S: State, M: Move> RandomAgent<S, M>
{
    pub fn new() -> RandomAgent<S, M>
    {
        Self
        {
            state_type: PhantomData,
            move_type: PhantomData,
        }
    }
```

```
}

impl<S: State, M: Move> Agent<S, M> for RandomAgent<S, M>
{
    fn recommend_move(&mut self, game: &mut Box<dyn Game<S, M>>, state: &S)
        ↪  -> M
    {
        crate::helpers::sample_and_consume(&mut rand::thread_rng(), game.
            ↪ get_moves(&state)).unwrap()
    }
}
```

From src/ai/game_runner.rs

```
use crate::ai::{agent::Agent, game::*};

pub fn run_game<S: State, M: Move>( mut game: Box<dyn Game<S, M>>, mut
    ↪ agents: Vec<Box<dyn Agent<S, M>>>, initial_state: S )
{
    let mut current_state = initial_state;
    let mut turn: usize = 1;
    println!("Board state:\n{}\n", current_state);

    while let None = game.result(&current_state)
    {
        turn ^= 1;
        let m = agents[turn].recommend_move( &mut game, &current_state);
        println!("Agent {} plays this move:\n{}\n", turn, m);

        // let m: nim::Move = agents[turn].recommend_move(&nim_state);
        current_state = game.perform_move(&current_state, &m);

        println!("Board state:\n{}\n", current_state);
    }

    println!("Winner: Agent {}", &turn);
}
```

BIBLIOGRAPHY

Beling, P. and Rogalski, M. (2020). On pruning search trees of impartial games. Artificial Intelligence, 283:103262.

Boros, E., Boros, E., Gurvich, V., Gurvich, V., Oudalov, V., and Oudalov, V. (2013). A polynomial algorithm for a two parameter extension of wythoff nim based on the perron–frobenius theory. International journal of game theory, 42(4):891–915.

Boros, E., Gurvich, V., Ho, N. B., and Makino, K. (2020). On the sprague–grundy function of extensions of proper nim. International journal of game theory.

Boros, E., Gurvich, V., Ho, N. B., Makino, K., and Mursic, P. (2017). Tetris hypergraphs and combinations of impartial games.

Boros, E., Gurvich, V., Ho, N. B., Makino, K., and Mursic, P. (2018). Sprague-grundy function of matroids and related hypergraphs.

Boros, E., Gurvich, V., Ho, N. B., Makino, K., and Mursic, P. (2019). Sprague–grundy function of symmetric hypergraphs. Journal of combinatorial theory. Series A, 165:176–186.

Bouton, C. L. (1901). Nim, a game with a complete mathematical theory. Annals of mathematics, 3(1/4):35–39.

Dufour, M. and Heubach, S. (2013). Circular nim games. The Electronic journal of combinatorics, 20(2).

Dufour, M. and Heubach, S. (2021). Circular nim cn(7,4).

Ehrenborg, R. and Steingrímsson, E. (1996). Playing nim on a simplicial complex. The Electronic journal of combinatorics, 3(1 R):1–33.

Grundy, P. (1939). Mathematics and games. Eureka, 2:6–9.

Horrocks, D. (2010). Winning positions in simplicial nim. The Electronic journal of combinatorics, 17(1):1–10.

Jenkyns, T. A. and Mayberry, J. P. (1980). The skeletion of an impartial game and the nim-function of moore's nim.sub.k. International journal of game theory, 9(1):51.

Larsson, U., Larsson, U., Rubinstein-Salzedo, S., and Rubinstein-Salzedo, S. (2016). Grundy values of fibonacci nim. International journal of game theory, 45(3):617–625.

López-Presa, J. L., Anta, A. F., and Chiroque, L. N. (2011). Conauto-2.0: Fast isomorphism testing and automorphism group computation.

Mursic, P. (2019). Hypergraph nim.

P. Codenotti, H. Katebi, K. A. S. and Markov, I. L. (2013). Conflict analysis and branching heuristics in the search for graph automorphisms.

Reading, N. (1999). Nim-regularity of graphs. The Electronic journal of combinatorics, 6(1):11.

Sprague, R. (1935). Über mathematische kampfspiele. Tohoku Mathematical Journal, First Series, 41:438–444.

Whinihan, M. J. (1963). Fibonacci nim. The Fibonacci Quarterly, 1(4):9–13.

VITA

Nelson Arthur Penn

Education

- B.S. in Mathematics from the University of Kentucky. Expected May, 2021.
- B.S. in Computer Science from the University of Kentucky. Expected May, 2021.

Professional positions held

- Summer 2018 and Summer 2019: IT Summer Intern at UPS.
- September 2018 to January 2020: Software Contractor at Connect, Inc.
- September 2019 to October 2020: Student Researcher at the University of Kentucky.

Scholastic and professional honors

- Patterson Scholar at the University of Kentucky.
- Dean's List: Fall 2017, Spring/Fall 2018, Spring/Fall 2019.