Nathan Fox
Math 640

Tools for Investigating Subtraction Games

1 Introduction

Everything that follows will be based on the following definition:

Definition 1. *A* subtraction game *is a counter-removing game where the legal moves consist of removing exactly some number of counters in a set S, called the* subtraction set*. The first player to be unable to move loses.*

One can consider subtraction games whose subtraction sets are either finite or infinite. We will only consider the finite case here.

We will also need the following definition:

Definition 2. *The* Sprague-Grundy function *of a position in a subtraction game is defined to be zero if the position has no legal moves, and it is the minimal excluded nonnegative integer (*mex*) of the Sprague-Grundy function values of all positions reachable from the current position otherwise.*

For a given subtraction set S, we define the *Sprague-Grundy sequence*, or *Nim sequence*, of S to be the sequence of values the Sprague-Grundy function takes on 0, 1, 2, etc. counters with subtraction set S.

It is well known that, if S is finite, then the Nim sequence of S is eventually periodic [1]. We will call the period of the period part of such a Nim sequence the *period* of the Nim sequence, and we will call the shortest possible prefix such that the remainder is periodic the *prefix* of the Nim sequence. Beyond this fact about periodicity, very little is known about general subtraction sets. The initial goal of this project was to discover some general facts about subtraction games. The eventual outcome is a Maple toolkit that can be used to examine specific sets of subtraction games in an attempt to learn information about them. In addition, it can be used to automatically write papers proving basic results about them. In Section 2, we discuss the implementation of these tools. In Section 3, we discuss how to use these tools.

2 Implementation

The core procedures in this package can be found in `code_finsubgames.txt`, which depends on `code_utilities.txt`, `code_dbfiles.txt`, `code_filters.txt`, and `code_selectors.txt`. The early procedures in this file compute Nim sequences and determine (with proof!) the prefixes (aperiodic parts), and periods of Nim sequences for finite subtraction sets. The way the code accomplishes this is by generating a large initial chunk of the Nim sequence for a given subtraction set S. It then looks for eventual periods of that chunk that repeat at least twice. For each such period, it then checks whether that period is a possible period for S. If it is a possible period, it then generates enough more of the Nim sequence necessary to confirm or refute that that eventual period persists. Once it has found a period satisfying all of these things, it tries to extend it as close to the beginning of the Nim sequence as possible. Finally, it returns the found period and prefix. In order to guarantee that it returns the simples period, it considers candidate periods in increasing order of length.

The next few procedures compute *contractions* of subtraction sets. A contraction of a subtraction set S is a subset $S' \subseteq S$ with the same Nim sequence as S. A criterion for identifying contractions is given in [1]. Another way of identifying contractions is by proving that S' and S have the same period and prefix. This is the method that this code uses, since all the framework has already been built for this.

The rest of the procedures are concerned with gathering large amounts of period and prefix data and analyzing these data. The first of these procedures format the data and write it to files in a format described in Subsection 2.1. The rest of the procedures work with these data files, either modifying them or analyzing the data they contain. These procedures make use of two abstractions, filters and selectors, which will be discussed in Subsection 2.2. All the data reporting procedures here return data in a raw format, which the user is free to manipulate as desired. Most of these procedures automatically open and reset the DB file passed to them. The procedures whose names contain `NoReset` do not reset the DB file. These procedures take an additional argument called `stopcond`, which is a filter. Searching begins wherever the file pointer left off, and searching stops once one satisfactory entry has been found and when stopcond fails to accept the next entry. Such procedures can be called repeatedly with different values of stopcond in order to only traverse a large file once, as opposed to once per call. Additionally, some of the procedures contain the word `Parallel` in their names. These procedures allow multiple searches to be done on a DB file simultaneously, even further reducing the number of file traversals required.

Finally, there is the file `code_paper.txt`. The first procedure in this file is `WritePaper`. This procedure uses its input to generate a human-readable paper from the raw data returned by procedures in `code_finsubgames.txt`. The format for this input will be discussed in Section 3. The second procedure is `ProvePeriodAndPrefix`. This procedure writes a theorem and proof structure for a given subtraction set, proving the prefix and eventual period of its Nim sequence. This procedure can be made to output either LaTeX or plain text. The third procedure, `WritePrefPerPaper`, allows for grouping a bunch of successive calls to `ProvePeriodAndPrefix`. The fourth procedure, `WriteSelfContainedPaper`, uses the third procedure in LaTeX mode to produce a file that can immediatley be compliled with LaTeX into a paper. The fifth procedure, `WriteSelfContainedPaperFromDBFile`, combines the filters of DB files with the paper writing capabilities of `WriteSelfContainedPaper` to allow for automatically generating lists of sets to analyze in a paper. The file `code_paper.txt` depends on `code_finsubgames.txt`.

2.1 DB Files

Procedures for manipulating files containing data about subtraction games, known as DB files, can be found in `code_dbfiles.txt`. An entry in such a file consists of at least six lines, in the following order:

1. A line beginning with S, followed by a space, followed by a space-delimited sequence of positive integers (possibly of length zero), followed by a newline. This sequence of numbers is the subtraction set for the entry.
2. A line beginning with F, followed by a space, followed by a space-delimited sequence of positive integers (possibly of length zero), followed by a newline. This sequence of numbers is the prefix for that subtraction set.
3. A line beginning with P, followed by a space, followed by a space-delimited sequence of positive integers, followed by a newline. This sequence of numbers is the period for that subtraction set.
4. A line beginning with G, followed by a space, followed by a positive integer, followed by a newline. This integer is the length of the prefix.
5. A line beginning with L, followed by a space, followed by a positive integer, followed by a newline. This integer is the length of the period.
6. One or more lines beginning with E, followed by a space, followed by a space-delimited sequence of positive integers (possibly of length zero), followed by a newline. Each such sequence of numbers is a contraction of the subtraction set.

Different entries in a DB file are separated by an extra newline.

Maple complains whenever an attempt to open an already-open file is made. Hence, `code_dbfiles.txt` includes wrappers for opening and closing DB files. Using the procedure `OpenDB` to open a DB file will never complain if the file was already open. Also, there are procedures for fetching the next entry or the previous entry. Entries are returned in the form `S,F,P,G,L,E`, where each of these terms is the corresponding entry component in its appropriate Maple data structure (set, list, list, integer, integer, set of sets). Finally, the procedure `ResetDB` makes it so that the next entry to be fetched is the first.

2.2 Filters and Selectors

Procedures related to filters can be found in `code_filters.txt`. A filter, at its most basic level, is a procedure with the following properties:

- It takes six arguments in the order `S,F,P,G,L,E`, which are the same data fields in the same formats as described in Subsection 2.1.
- If the inputs are all of the proper types, it returns a boolean value.

- If the input `S` is passed as 0 (as opposed to as a set), then it returns a string.

More to the point, the boolean value is `true` if the filter "accepts" the entry, and it is `false` if the filter "rejects" the entry. Also, the string is a description of what the filter checks for. If the user wishes to use a custom filter with the WritePaper procedure, this description should begin with a present progressive of a verb, and the phrase should be able to function as an adjectival phrase in a sentence.

The file `code_filters.txt` provides four groups of procedures for manipulating filters. The first is a collection of procedures that take filters as arguments (possibly along with other arguments), and they return new filters based on those filters. For example, the procedure `AndFilters` returns the boolean conjunction of two filters, and it modifies the description appropriately. The second group is a collection of procedures which take non-filter arguments and return filters based on those arguments. For example, the procedure `MakeSizeNFilter` takes an argument `n` and returns a filter that checks whether `S` has cardinality exactly `n`. The third group is a collection of procedures which take no arguments and return "mutable filters". These filters have some memory and may not always return the same value on the same input. For example, the procedure `MakeStrictlyLongestPrefixSoFarFilter` returns a filter that returns `true` if and only if the argument `G` is greater than any other value of `G` this filter has seen before. The final group is a collection of premade filters. For example, the procedure `IdentityFilter` is a filter that always returns `true`.

Procedures related to selectors can be found in `code_selectors.txt`. A selector, at its most basic level, is a procedure with the following properties:

- It takes six arguments in the order `S,F,P,G,L,E`, which are the same data fields in the same formats as described in Subsection 2.1.
- If the inputs are all of the proper types, it returns something.
- If the input `S` is passed as 0 (as opposed to as a set), then it returns a string.

More to the point, the "normal" return value is whatever the selector is "selecting" from the input. This is usually some part of the input, possibly in a modified format. The string return value is a description of what the selector does. If the user wishes to use a custom selector with the extract capabilities of the WritePaper procedure (see Section 3 for more information on this), this description should be a noun phrase.

The file `code_selectors.txt` provides three groups of procedures for manipulating selectors. The first is a collection of procedures that take selectors as arguments (possibly along with other arguments), and they return new selectors based on those selectors. For example, the procedure `AddSelectors` returns the numerical sum of two selectors, and it modifies the description appropriately. The second group is a collection of procedures which take non-selector arguments and return selectors based on those arguments. For example, the procedure `MakeConstantSelector` takes an argument `n` and returns a selector that always returns `n`. The final group is a collection of premade selectors. For example, the procedure `SSelector` is a selector that returns `S`.

Ideally, filters and selectors would be implemented as objects with an intricate inheritance pattern (they would both extend the same class, and each different filter/selector would be a subclass). Though Maple supports objects, they are just glorified modules and do not support inheritance. Hence, filters and selectors have been implemented as procedures in the manner described.

3 Interface

3.1 WritePaper

First, we will discuss in detail how to use the `WritePaper` procedure. The raw procedures in `code_finsubgames.txt` use similar arguments, but are less complicated. The procedure `WritePaper` takes five arguments:

1. `outfile`: the name of the file to which the paper should be written
2. `files`: a list of DB files to process. The files will be processed in the order given.
3. `descs`: a list of strings. This list has the same number of entries as `files`, and the entries in `descs` are descriptions of the corresponding files in `files`. If no description of the file is desired to appear in the paper, its description should be included as `false`.
4. `argseq`: a nested list data structure that is the meat of the procedure. Details are described below.
5. `opts`: a set of options, each of which is passed as a string. Right now, there is only one option the procedure recognizes: "FLOAT". If included, averages are reported as floating-point numbers. If excluded, averages are reported as fractions.

There are three different types of reports that can be performed and written into a paper. The first is a count. A count goes through a file and counts the number of entries satisfying a given filter. The second type of report is an average. An average uses a filter and a selector that returns a numerical value. It goes through a file and takes the average of the selector's value over all entries satisfying the filter. The third type of report is an extract. An extract also uses a filter and a selector, though the selector can have any return type. It goes through a file and gathers in a list the selector's values over all entries satisfying the filter. The procedure `PaperWriter` converts all of these report types into human-readable English.

The argument `argseq` is a list of lists of lists. The outermost list contains the same number of entries as `files`. Each of these entries (which are lists of lists) contains six-element lists of the form `[countfilters, avgfilters, avgselectors, extfilters, extselectors, stopcond]`. These correspond to a collection reports to run on the file. The following is an explanation of each of these parameters:

- `countfilters`: a list of filters. Each filter in this list will be used to generate a count report. If no count reports are desired, this list can be empty.
- `avgfilters`: a list of filters. Each filter in this list will be used to generate an average report. If no average reports are desired, this list can be empty.
- `avgselectors`: a list of selectors with the same number of entries as `avgfilters`. This list will be used in conjunction with `avgfilters` to generate the average reports.
- `extfilters`: a list of filters. Each filter in this list will be used to generate an extract report. If no extract reports are desired, this list can be empty.
- `extselectors`: a list of selectors with the same number of entries as `extfilters`. This list will be used in conjunction with `extfilters` to generate the extract reports.
- `stopcond`: a filter. All report generators will stop running when this filter fails to accept the current entry, provided that at least one report filter has accepted something thus far.

The file will not be reset before the next set of reports runs. To reset it, pass it multiple times in the files list.

3.2 Self-Contained Papers

Now, we will describe how to use the `WriteSelfContainedPaperFromDBFile` procedure. During this discussion, we will also explain the remaining procedures in `code_paper.txt`. The procedure `WriteSelfContainedPaperFromDBFile` takes five required arguments and two optional arguments. They are listed and described here, in order.

`filter`: This argument specifies a filter to apply to a DB file to select subtraction sets to put into the paper. This argument is required.

`infile`: This argument is the name of the DB file to which to apply the filter. This argument is required.

`outfile`: This argument is the name of the file to which to output the paper (should have extension .tex). This argument is required.

`title`: This argument is the title of the paper. This argument is required.

`author`: This argument is the author of the paper. This argument is required.

`opts`: This argument is the set of options to use when writing the paper. See the section on options below for a description of how to use this argument. This argument is optional. Omitting it is equivalent to passing the empty set.

`stopcond`: This argument has no effect unless the NORESET option is used (see the section on options below). In that case, it specifies a filter that indicates when searching for sets in the DB file should stop. Specifically, searching will stop when at least one set has been found and when `stopcond` fails to be satisfied. This argument is optional. Omitting it is equivalent to passing `false`, which has the effect of only stopping when the whole file has been scanned.

The procedure itself will print nothing, and it will output a LaTeX file to `outfile` that can then be compiled into a PDF. It works by using the filter on the DB file to create a list of subtraction sets, which it then passes with all of its other arguments to the procedure `WriteSelfContainedPaper`. This procedure has the same type of input and output as `WriteSelfContainedPaperFromDBFile`, except that it takes a list of subtraction sets as a first argument in place of a filter and a DB file. Also, its last (optional) argument is a string, `descr`, as opposed to a filter. If provided, this string should provide a short description of what the sets in the paper have in common in the same grammatical form as a filter description. The procedure `WriteSelfContainedPaper` calls `WritePrefPerPaper` to write the bulk of the paper, which, in turn, calls
`ProvePeriodAndPrefix` once to produce a proof for each set in the list. All four of these procedures can take a number of options, which we will discuss next.

3.2.1 Options

The procedures `ProvePeriodAndPrefix`, `WritePrefPerPaper`, `WriteSelfContainedPaper`, and `WriteSelfContainedPaperFromDBFile` all have an optional argument `opts`. This argument should be a set of double-quoted strings. Omitting the opts argument is equivalent to passing the empty set. What follows is an alphabetized list of all meaningful options, which procedures they apply to, and what they do. Passing an option not in this list has no effect, unless it contains one of these option names as a prefix (in which case the behavior is undefined). Additionally, remember that all of these options must be passed as double-quoted strings (i.e. “DETAILED”).

DETAILED:

Applies to: `ProvePeriodAndPrefix`, `WritePrefPerPaper`, `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: Replaces abbreviations (e.g. “mex”) with their non-abbreviated forms (e.g. “minimum excluded element”).

DISPLAY:

Applies to: `ProvePeriodAndPrefix`, `WritePrefPerPaper`, `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: No effect unless LATEX option also present (or LATEX mode automatically employed). In that case, pieces of Nim sequences are put into display mode whenever they appear.

INSTRUCTIVE:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, not all sets in the list will have proofs written for them. Rather, a random selection will have proofs written. This number is, by default, proportional to the logarithm of the number of sets. One of the sets proved will the a set with the longest period, and another set proved will be the set with the longest prefix (if there is a nonzero prefix and this is not the same set as the longest period). In addition, one of the sets will always be one without a prefix (if possible). Attaching a number to the end of the word "INSTRUCTIVE" (i.e. "INSTRUCTIVE3") is allowed. This will force the number of proofs to equal that number.

INTRO:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, includes an introduction in the paper. If `WriteSelfContainedPaper` is called with a string passed for the argument `descr`, then that description is used as a part of the intro. The procedure `WriteSelfContainedPaperFromDBFile` seeds that description from the description of its filter. Other than the sentence or two stemming from that description, all intros are the same. Including an introduction also includes a bibliography at the end of the paper, as the generic intro includes a citation.

LATEX:

Applies to: `ProvePeriodAndPrefix`, `WritePrefPerPaper`

Effect: If present, the theorms and proofs will be produced in LaTeX code, assuming the presence of the `amsthm` package and appropriate `\newthm`s. Procedures `WriteSelfContainedPaper` and `WriteSelfContainedPaperFromDBFile` automatically produce LaTeX code, so this option need not be present.

NICEHBOX:

Applies to: `ProvePeriodAndPrefix`, `WritePrefPerPaper`, `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: No effect unless LATEX option also present (or LATEX mode automatically employed). In that case, pieces of Nim sequences in theorms and proofs are made multi-line if they get to long in an attempt to avoid overfull hboxes. Attaching a number to the end of the word "NICEHBOX" (i.e. "NICEHBOX75") is allowed. This will force the number of characters per line of sequence text to be close to that number.

NODATE:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, today's date will not appear with the title and author of the paper.

NORESET:

Applies to: `WriteSelfContainedPaperFromDBFile`

Effect: If present, the DB file `infile` will not be reset before being scanned for sets matching the filter. The argument `stopcond` will only be taken into consideration if the NORESET option is present.

RANDOM:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, not all sets in the list will have proofs written for them. Rather, a random selection will have proofs written. This number is, by default, proportional to the logarithm of the number of sets. Attaching a number to the end of the word "RANDOM" (i.e. "RANDOM3") is allowed. This will force the number of proofs to equal that number. If both INSTRUCTIVE and RANDOM are present, INSTRUCTIVE takes precedence. If exactly one of these defines a number, that number will be used. If either of these appears more than once, or if more than one number is defined, the behavior is undefined.

SORTED:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, sets will be considered in colexicographic order (treated as if their elements are in increasing order). If absent, sets will be considered in the order they appear. This argument applies to both the summary and the proof order. If either RANDOM or INSTRUCTIVE is present, then it only applies to the summary, as the proofs will be sorted automatically.

STRING:

Applies to: `WritePrefPerPaper`, `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: Instead of writing the result to a file, return it as a string. The argument `outfile` is then ignored.

SUMMARY:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: If present, includes a summary table of the subtraction sets, periods, and prefixes in the paper.

SUMMNICEHBOX:

Applies to: `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: Prefixes and periods in the summary table are made multi-line if they get to long in an attempt to avoid overfull hboxes. Attaching a number to the end of the word “SUMMNICEHBOX” (i.e. “SUMMNICEHBOX25”) is allowed. This will force the number of characters per line of sequence text to be close to that number.

VERBOSE:

Applies to: `ProvePeriodAndPrefix`, `WritePrefPerPaper`, `WriteSelfContainedPaper`, `WriteSelfContainedPaperFromDBFile`

Effect: Adds a lot of detail to proofs. Instead of the proof length being

$$O\left(\text{period length} + \text{prefix length} + \text{constant}\right),$$

it is now

$$O\left(\left(\text{period length} + \text{prefix length}\right) \cdot \text{constant}\right).$$

References

[1] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Vol. 1, A. K. Peters, 2001.