SPADE: A Library for Programmatic Parsing and Verification of Discrete Data Structures

1st Russell Phillips
Computer Science
Idaho State University
Pocatello, USA
russellphillips@isu.edu

2nd Paul Bodily
Computer Science
Idaho State University
Pocatello, USA
bodipaul@isu.edu

***Abstract*—SPADE is a library written in C# that takes string instances of discrete math data structures and parses them into data structures that are easily accessible using C#'s built in data structures. It also checks that the string instances appropriately follow specified constraints, that the various sets and lists of a problem relate to each other in a specified way. These constraints are defined using a language written as a string that closely follows normal discrete math syntax. Using this language, SPADE provides a flexible framework for defining problem types out of basic discrete math operations, which allows efficient parsing and validation of string representations of that problem.**

***Index Terms*—Discrete math, string, parsing, verification, access, C# library**

I. Introduction

There are a wide variety of computer science problems that are defined by the language of discrete math. While programming languages often support many of the operations discrete math uses, the actual programming syntax of these operations are very different than what is typically seen in actual discrete math. While it would be hard to design a system that uses the exact language discrete math usually uses, a system that uses much of discrete math language is possible. This is because many of these problems are defined with elementary items, like sets, lists, and functions, along with the relations these sets and lists should have with each other. Therefore, a system that understands these relations can take a definition for these relations and be able to parse arbitrary discrete math problems, all within software.

An example of a problem defined by discrete math is Set Cover. Given a universal set U, and a set S such that $\forall a \in S, a \subseteq U$ find the smallest subset of S, C where the union of all elements of C is U. This problem can be applied to some real world problems, for example planning shifts for personnel [6]. The Universal set is every shift that needs to be covered, while the subsets in S are the possible schedules for an employee following constraints like 2 days off in a week, or 16-hour breaks before the next shift. Then, the Set Cover of this is the minimum number of employees that will be needed to cover all shifts and their schedules (see Fig. 1). The smallest set cover in this case is hiring employee 1 and 2, to cover the three shifts on Monday, Tuesday, and Wednesday.

Set Cover can model real world problems, while being defined in the language of discrete math. Further, it uses only a small number of elements for its definition, namely sets, for each, subsets, and unions.

	Monday	**Tuesday**	**Wednesday**
Employee 1	✓	✓	×
Employee 2	×	×	✓
Employee 3	✓	×	×

Fig. 1. An example set cover problem instance representing employee availability. Here the universal set $U = \{Monday, Tuesday, Wednesday\}$ and $S = \{\{Monday, Tuesday\}, \{Wednesday\}, \{Monday\}\}$. A solution to this instance would represent the minimum number of employees needed to cover all shifts.

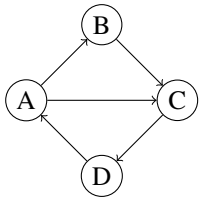

Fig. 2. The SPADE library can parse arbitrary data structures using common discrete math notation. For example, consider that the string representation for a graph is commonly defined using an ordered pair (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. Thus, to parse graphs using SPADE, a parsing object is first seeded with this definition, following which the parser object is then able to parse, validate, and facilitate access to the subelements of arbitrary graph instances. Here $N = \{A, B, C, D\}$ and $E = \{(A, B), (B, C), (C, D), (D, A)\}$

Graph based problems are also very common in discrete math. Graphs can be represented as a pair of a set of nodes, and a set of edges, which are pairs of nodes - see Fig. 2 for an example of a graph represented this way. Nesting these structures are also possible, for example Clique, another NP-complete problem to find K nodes in a graph that are all interconnected. Clique can be represented as a pair of the graph and an integer K.

Another example is Steiner tree, where given a graph and a set of terminal nodes, and an integer K, the task is to find a subtree of size K or less that contain all the terminal nodes. This can be represented as a 3-pair or list (G, T, K) where T is a subset of the nodes of G.

All these problems use only a handful of concepts from discrete math, namely sets, lists, for each, subsets, or is an element of. A parser that can only understand these things is

already very powerful.

Structured Parsing and Analysis for Discrete Elements or SPADE is a library written in C# that takes string instances of problems and parses them into data structures that are easily accessible using C#'s built in data structures. Code and instructions on how to install can be found here[1] SPADE also checks that the string instance appropriately follows the constraints of the problem. For example, in the set cover problem it is capable of enforcing that every set in S is indeed a subset of the universal set. It then gives access to two variables named U and S that allow access to sets and subsets of that problem. The formal language that defines each problem in SPADE is designed to be close to the standard notation of discrete math while still easy to program with and implement.

II. Related Works

Research has been done on how concretizing discrete math paradigms through computer programming modules amplifies student understanding of discrete math concepts. In one study, an experimental group of discrete math students were given programming assignments that employed Python syntax extensions to represent "for all" and "there exists" quantifiers. These students were shown to have elevated test scores compared to the control group students who were not given these same assignments [3]. In another paper, researchers designed coursework which has students model logic puzzles such as Sudoku and Numbrix, by representing them as predicate logic satisfiability problems accepted by 3rd-party satisfiability modulo theories (SMT) solvers. Their goal was to have students learn the difference between a problem instance, and a problems solution. [1]. Others have found that when Python is used to introduce discrete math concepts by first showing examples in code that the students can run, and then connecting code to corresponding math notation, students show increased grades and retention. [4]. Many programming languages (e.g., *Mathematica* [2]) support discrete math notation, however, they often define their own syntax when using these discrete math concepts, instead of more closely following standard mathematical notation. SPADE aims to bridge this gap and provide syntax more similar to standard mathematical notation.

Human readable text is also an important part of programming and discrete math. For example, text2math is a model for parsing mathematical word problems into math expressions that can be used to solve those problems. [7]. Redux is a knowledge base of NP-complete problems and their reductions designed to teach students using interactive visualizations of those problems and their reductions. Problems in Redux's knowledge base are represented using strings, since a flexible way for representing many types of problems while allowing users to input their own problem instances [5]. SPADE is especially designed for this type of use case, to help turn string instances of problems into useful data structures for further use in reductions or solvers.

[1]https://github.com/Jetison333/SPADE

III. Methods

To use SPADE, it is first provided with a *definition string*. This definition string contains the information for how a particular problem is set up, and contains the same information as the formal definition of a problem. This definition string is written in its own language that is central to SPADE. It is represented using only ASCII characters so it is easily typed, and closely follows discrete math notation. Then, SPADE is provided with an *instance string*, a specific instance of that problem as represented as a string. An instance string is one problem out of the set of all possible problems of that problem type, which contains what the contents of each element of a problem. A definition string tells SPADE the problem type, and an instance string is a specific instance of that problem type.

A. Instance strings

Since instance strings represent an individual instance of a problem, they must have a few components. The highest level is a *collection*, and SPADE makes a distinction of collections between sets and lists, where sets are unordered groupings and lists are ordered groupings. Collections contain *elements*, which may be another collection, or a basic object like a integer or word. Particular instance strings are typically nested, for example set cover being a *pair* (a list of length 2) that contains a set of objects and a set of sets of objects.

Collections are represented within an instance string as strings. We represent a set U of elements $e_0, ..., e_k$ using curly braces as follows

$$U = \{e_0, ..., e_k\}$$

So a set may look something like $\{a, b, c\}$. A list V, will be represented using parenthesis as follows

$$V = (e_0, ..., e_k)$$

as such, an example of a list could look something like $(1, 2, 3)$. Also note that since an element of a set or list may also be a set of a list, so a set of lists can be represented as $\{(1, 2), (2, 3)\}$

B. Definition String

SPADE models a new problem as defined by a *definition string*. This string allows the user to specify how an instance of the problem is to be formatted, and the constraints that problem should follow. This follows a set builder notation, such that the set builder describes the set of every possible instance string of that problem. SPADE is equipped to parse and validate that instance strings are in the set as defined by the definition string. Consider the example of set cover, which a possible definition string is as follows

```
{(U,S) | U is set,
S subset {a | a subset U}}
```

This is a description of set cover as a pair, where the first element in the pair is the universal set U, and the second element is a set that is defined as a subset of all possible

subsets of U, which is an equivalent way to phrase that every element of S should be a subset of U. This is a version of the formal definition of set cover.

Since SPADE implements its own language it was necessary to build a parser for that language. For this, ANTLR, a tool that generates parsers was used to generate a visitor pattern parser. The definition string is made out of the following parts:

1) Format String: In the definition string, first there is a *format string*, which is the part to the left of "|". The format string represents the high-level structure of a problem, and gives names to the different parts of it. This uses the same convention as the problem instance strings do, with sets being represented by curly braces and lists with parentheses. Then, each element inside the collection is given a name, which is used in the later parts of the definition string. In the set cover example, (U, S) means that set cover is a list of size 2 (i.e., an ordered pair) where the first element is called U and the second is called S, where U and S each contain the specific elements of a particular problem instance.

2) Constraints: After the keyword "|" is a list of assertions or *constraints* separated by commas that are assumed to be true for a valid problem instance. The language used for defining constraints in SPADE is modeled after the language of Set theory. Each constraint says that some named element is either a basic object, or related to the other objects in some way. "U is set" simply means that U should match the pattern of a set, namely curly braces around a list of elements. "is" can also be followed by "list", or "int" which would constrain U to be a list or to be some integer respectively.

Named elements can also be related to each other by using elementary set operations. The term "subset" requires that the left-hand side must be a subset of the right-hand side. In the example, the right-hand side uses a set builder notation, as SPADE allows nesting of set builders. Just as with the overall set builder, there is a format string, and then a list of constraints. The English version of this example is "a, for every a that is a subset of U", or "all possible subsets of U". This set builder generates a set of all possible a's that match the constraint "a subset U". Since S is a subset of this set, this means that S must be some subset of all possible subsets of U, which matches the description of set cover.

Other possible ways to construct objects are with binary operations; operations that take two other objects as input. These include "union", "intersection", and "cross". These included with the set builder notation allow for a parsers to be defined for a variety of problems.

C. Implementation

As an example, this is what code for setting up set cover in SPADE would look like.

```
StringParser set_cover = new SPADE(
    "{(U,S) | U is set, \
      S subset {a | a subset U}}")
set_cover.parse("({1,2,3}, \
                 {{1,2},{2,3},{1,2,3}})")
UtilCollection U = set_cover["U"]
UtilCollection S = set_cover["S"]
```

First, SPADE is given the definition string for set cover. It compiles this definition string internally, and then waits for a instance string.

1) UtilCollection Class: The first thing that happens to an instance string is that it is turned into a UtilCollection class object. UtilCollection is a class that combines unordered and ordered sets, since SPADE deals with both of these. An object UtilCollection can act as either an ordered or unordered collection, or as a string. This class is also recursively defined, meaning every element inside of a UtilCollections object is also a UtilCollections object. This allows arbitrary sets and lists of strings to be defined inside of one UtilCollections object. This class also has a constructor that facilitates instantiating of UtilCollections objects directly from instance strings. The class has methods for indexing, checking membership, iterating over its contents, and creating new objects with set operations.

In the example, the instance string is turned into a UtilCollections object that represents a pair, of which the first element is a set that contains 1, 2 and 3, and the second element is a set of sets.

2) Format: Next, the UtilCollections object is matched to the format string. When matching the format string with a particular instance, SPADE does a recursive walk, either breaking up each UtilCollection for the next step or adding each piece to a dictionary, with the key being the name of that element. In this example, the UtilCollections objects first element is assigned the name "U", and the second is assigned to the name "S".

3) Constraints: Now that the names have been matched to some element, the list of constraints can be applied to those elements. For each constraint, SPADE builds functions from the bottom up that describe each piece of a constraint. For describing collections, there are two different cases that SPADE tracks, the main one where the function checks if an element is part of a set or a list, and a secondary one generates the whole set or list. This is done because often its computationally expensive to generate the entirety of a set that is required for a constraint.

For example, in "{a | a subset U}" its not needed to generate every possible subset of U, just make a function that checks if something is a subset of U.

These functions are built from the ground up where the more complicated cases are built from the simpler cases. For example, if a constraint contains "A union B" then the member checking function can be built out of the member checking functions for A and B, by checking if the input is a member of either A or B, which is the union of A and B. Other operations are done similarly.

4) set builder notation: Set builder notation is done by using the format methods and the constraint methods. Similarly to constraints, its capable of generating the set the set builder notation describes, or checking if an element is part of that

TABLE I
BIG O OF SUPPORTED OPERATIONS

	generate big O	check big O
Union	n	1
Intersection	n	1
Cross	n^2	1

set. To check if an object is part of the set, it first uses the format method to break it into its parts, and then checks to see if the constraints hold. This is the same process done at the top level to validate the overall problem instance.

5) output: Once every constraint is checked, SPADE provides access to the dictionary containing each named collection, which can be used in whatever way is needed.

IV. RESULTS

SPADE has already been used in Redux [5] to parse one of the twenty four existing problems, and is capable of parsing at least twenty more of the existing problems.

Since SPADE follows a recursive structure its possible to make new relationships that are not already included in SPADE. For example, one way to represent a directed graph would be the following

```
{(N,E} | N is set, E subset N cross N}
```

This represents a graph as an ordered pair of nodes and edges. The nodes are simply a set of nodes. The edges are represented as a set of ordered pairs of nodes, which is an edge going from the first node of the pair to the second. "cross" is a binary operation that does a cross product, which is the set of every possible pair of elements. However, even without the specific operation "cross" SPADE would still be able to represent a graph with the following definition string

```
{N,E} | N is set, E subset
{(a,b) | a in N, b in N}}
```

This achieves the same structure and constraints as the previous, without the explicit use of "cross". It checks that every element of E is an ordered pair, where the first element is in N, and the second element is also in N, which is the same as the definition for cross-product. Another example is the operation set difference, which is written $A \setminus B$, and the result is every element in A that is not in B. Despite not directly existing in SPADE it can still be represented as the following.

```
{x | x in A, not x in B}
```

A. Performance Analysis

Since every element of a set does not need to be created to check most constraints, checking that an element is in a set is much more efficient than generating the entire set. Table I lists the big O of the three main binary operations, where n is the length of the two sublists.

In the case of the "in" keyword, SPADE must only check that one collection is in another one. Since checking an element is constant time per binary operation, the overall big O only depends on the number of binary operations used, which for most problems will be fairly low. For subsets, SPADE must go through the previous process n times, where n is the size of the collection on the left since it checks that each element exists in the right-hand set. These types of constraints are still fairly efficient, however, the constraint "is" is forced to generate the entire set to check overall equality, which could be quite large, particularly in the case of cross-products.

V. DISCUSSION

A possibility for future work for SPADE is to implement the universal quantifiers "for all" and "there exists". Currently, while SPADE can check many kinds of constraints, it is not possible to do all kinds. For example, any problem whose definition contains a function it is not possible to completely constrain that problem. Given a definition of a function $F : A \to B$ where F is a subset of $\{(a, b) \mid a \in A, b \in B\}$, for all $a \in A$, there exists $b \in B$ such that $(a, b) \in F$, and $|F| = |A|$. The first constraint, that each element in F is a pair of an element from A and an element from B, is possible to represent in SPADE, but the second, that there is one and only one pair containing each element from A is not.

Another potential would be a way to define operations and structures for use in other definition strings. For example, many types of problems are based on graphs, but they often have extra parameters that are needed for the complete definition (like K for Clique and a set of terminal nodes for Stiener tree). For this reason it would be useful to be able to define a graph (N, E) with its list of constraints, and be able to reuse that definition in multiple places. In a similar way, it also would be useful to be able to define operations, for example the previously discussed set difference for reuse in multiple places.

Another avenue to explore is extending the language to allow finding solutions to problems. Currently SPADE only checks that a string instance is a possible instance of the problem as defined by the definition string, without any mention of what the solution to a problem looks like. It may be possible to extend SPADE with the capability to solve problems which would necessitate operations like finding the minimum or maximum of some variable.

VI. CONCLUSION

SPADE provides a structured approach for parsing and verifying discrete math problem instances, enabling developers to work with discrete math structures in a programmatic manner. After being provided with a definition string, SPADE ensures that an instance string adheres to well-defined constraints. The library's use of basic discrete math concepts and systems build out of those concepts, allows it to support a wide range of discrete math problems, from Set Cover to graph-based problems like Clique and Steiner Tree. With its syntax that follows discrete math notation while still being representable by ASCII, SPADE bridges the gap between

theoretical discrete math and practical software implementation. Future work includes expanding support for additional mathematical constructs, allowing user-defined constructs, and integrating SPADE with other computational tools.

REFERENCES

[1] Shin Hong. Using smt solver and logic puzzles for teaching computational logics in discrete mathematics class. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1381, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Wolfram Research, Inc. Mathematica, Version 14.2. Champaign, IL, 2024.

[3] Yanhong A. Liu and Matthew Castellana. Discrete math with programming: A principled approach. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1156–1162, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Marcelo Malheiros and Claus Haetinger. Explorable discrete mathematics: a python-based undergraduate-level teaching approach. In *Anais do XXXV Simpósio Brasileiro de Informática na Educação*, pages 2494–2505, Porto Alegre, RS, Brasil, 2024. SBC.

[5] Kaden Marchetti, Andrija Sevaljevic, Alex Diviney, Caleb Eardley, Russell Phillips, Rajiv Khadka, Daniel Igbokwe, and Paul Bodily. Redux: An interactive, dynamic knowledge base for teaching np-completeness. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 255–261, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Fabio Schoen. 20. set covering, packing, partition¶, Apr 2024.

[7] Yanyan Zou and Wei Lu. Text2math: End-to-end parsing text into math expressions, 2019.