Congruence Closure Algorithm

Report on the Automatic Reasoning project

Chiara Solito,
VR487795

June 2023

1 Introduction

This project aims to implement the congruence closure algorithm with DAG to satisfy a set of equalities and inequalities in the quantifier-free fragment of the theory of equality. In this project, the algorithm was implemented with a variation: **non-arbitrary choice of the representative of the new class in the union function.**

2 Delivery

The program is delivered via a GitHub repository. The whole structure of the repository was also sent in the form of a compressed archive to the instructor. To execute it, you have to:

1. Clone the repository, or extract the files from the zip folder.
2. Copy, or drag and drop, in the folder `data` the test file you want to input to the program. Please be careful with the extension of the file and the format.
   More information on how to format the accepted input can be found in ***subsection 4.1*** or in the `README` of the repository.
3. If it's the first time you run the algorithm, you have to run the bash script `run.sh`: the bash script will create and activate a venv environment, and automatically install each dependency of the project. It will also start the program and ask for the input file name (with its extension, e.g. `input.txt`).
   After the first time, you can simply call the main script:

```
source ./venv/bin/activate
python3 main.py
```

3 Development

The project was implemented using `python3` in version 3.11. The structure of the project is as follows:

- **main** folder: contains the `main.py` script, the run.sh bash file, the README, and the requirements text file, plus all the other folders.
- **src** folder: contains the principal components of the project, the `dag.py` script, containing DAG class, and both *parsers*.
- **data** folder: contains the input already delivered with the project (results from this input and analysis can be found in ***section 5***

3.1 Main Data Structures

Directed Acyclic Graph The main class and center of the project is the DAG class, implemented in the `dag.py` script.
The main attribute of the class is a graph structure `DiGraph` from the library Networkx: this is the base class for directed graphs. A DiGraph stores nodes and edges with optional data, or attributes.
The DAG class also stores two lists: equalities and inequalities.

```
def __init__(self):
    self.g = nx.DiGraph()
    self.equalities = []
    self.inequalities = []
```

It also has all the functions that are needed for the computation of the algorithm:

- `add_node`: it's a function that adds a node, with a unique id (a UUID4), an fn (so a symbol), the node arguments, the node "finds" and the node congruence parents. When a node is instantiated, of course, the find will point to itself, while the 'cc_par' attribute will be empty.
- `add_edge`: when an edge is added between two nodes while constructing the graph, the parent node will be added to the list of parents of

such node, while the child node will be added to the list of arguments of the parent. In an opposite fashion `remove_edge` (that gets used when compacting the graph structure) removes both the node from each other lists.

- `node()`, `find()`, `ccpar()`, `congruent()`, `merge()` are the exact implementation of the pseudocode introduced both in class and Sect. 9.3 of the Bradley-Manna textbook [1], with the exception of the `union()` function, where we implement the non-arbitrary choice of the representative of the new class, picking the one with the largest ccpar set.

- `simplify`: as we will elaborate while talking about the Parser, the graph, as it is created in the beginning, repeats each node all the times it encounters it. With the simplify function each leaf with the same symbol gets unified, and so is each equivalent function (e.g. f(a,b) and f(a,b) gets unified.)

Parser The parser takes in input everything that is in the text files, or a string, passed via the smt parser. Each element of the list is an equality or an inequality. Each equation gets divided by the symbol: if it's an equality the couple of clauses will be assigned as a new item of the equalities list, while if it's an inequality, the couple will be in the other list. Each literal gets parsed as a list of lists, in which the brackets are the delimitation of each sublist: $f(f(a))$ gets parsed as $[f, [f, [a]]]$. For each list, if it encounters a single item, the parser will create a node, if it encounters another list, will treat the last node added as a parent and the list as its sons.

SmtParser The smt parser uses the class `SmtLibParser` from the library PySmt. Based on what it returns we readapt the string to be subsequently parsed by the principal parser.

3.2 Main

The main function elaborates the input and then gives it to the parser. If the input is in the smt2 format, first it will be parsed by smt parser and then by the main parser.
The parser takes in input the empty graph structure and then builds it, when function `parse_formula` gets called. If for example we give to the parser:

$$f(a, b) = a \wedge f(f(a, b), b) \neq a$$

the first instance of the graph generated, and given in output will be as in Figure 1.

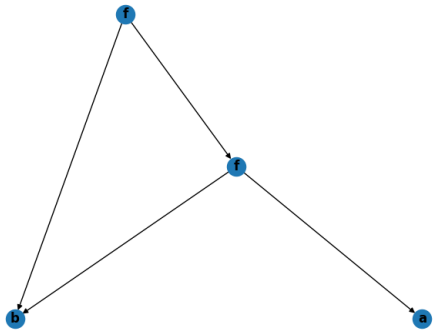

Figure 1: DAG representation.

For each couple in the equation list we call the merge function, then for each inequality we check the find of each literal in the couple. If we find an inequality of the form $t_1 \neq s_1$ for which $t_1.find() = s_1.find()$ we can say that the formula is unsatisfiable.
The final graph for the formula we've seen before will be as in Figure 2. This formula is unsatisfiable.

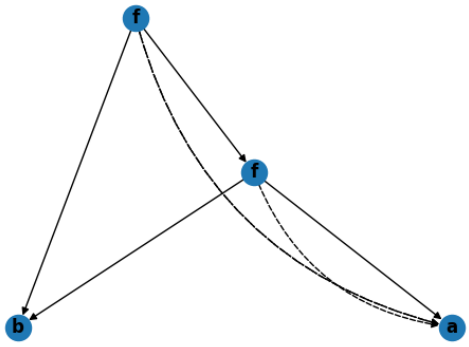

Figure 2: figure
DAG representation with find.

4 Accepted Input

4.1 Text files

Files with `.txt`' extension are accepted if they are presented in AND with each other. Each equality and disequality to be in consideration in the algorithm must be put on different lines.

Example of the structure of an 'input.txt' file:

```
f(a,b) = a
f(a,b) != a
```

This will be interpreted as '$f(a, b) = a \wedge f(a, b) != a$'

4.2 SMT files

Smt2 format is accepted only when formulas are in AND with each other. As for the text files, the formula is read as different equalities and disequalities in AND.

5 Test and Results

The script was tested on 14 input files. Nine are in the txt form and 5 in the smt2 form, you can find all files in the folder `data`. All tests are correct: a summary of tests is visible in Table 1.

6 Critic Analysis

Smt Files The principal problem that can be encountered is when using `.smt2` files. To put the input in a formula compatible with the one accepted by the parser written, we play with the string: sometimes this creates problems with brackets and makes the process of the input impossible.

DNF and CNF formulas The main limitation, of course, is encountered with formulas that aren't in AND, since we don't accept OR, quantifier, etc. in the formulas.

Input	True result	Result of the algorithm	Exec Time
input1.txt	UNSAT	UNSAT	0.00025 s
input2.txt	SAT	SAT	0.00033 s
input3.txt	UNSAT	UNSAT	0.00022 s
input4.txt	UNSAT	UNSAT	0.00041 s
input5.txt	SAT	SAT	0.00097 s
input6.txt	UNSAT	UNSAT	0.00067 s
input7.txt	SAT	SAT	0.00020 s
input8.txt	SAT	SAT	0.00070 s
input9.txt	UNSAT	UNSAT	0.00073 s
input1.smt2	UNSAT	UNSAT	0.00023 s
input2.smt2	UNSAT	UNSAT	0.00089 s
input3.smt2	UNSAT	UNSAT	0.00047 s
input4.smt2	UNSAT	UNSAT	0.00024 s
input5.smt2	UNSAT	UNSAT	0.00026 s

Table 1: Results of tests

References

[1] Aaron R. Bradley, Zohar Manna, *The Calculus of Computation, Decision Procedures with Applications to Verification*, 2007 - Springer Science and Business Media.