Congruence Closure Solver

Project Report 2024-2025

Luca Panariello VR518122

1 Introduction

The project consists of creating a **solver** for the satisfiability of formulas belonging to three different theories: the theory of equality with free symbols, non-empty possibly cyclic lists, and arrays without extensionality.

The solver uses the **Congruence Closure Algorithm** on a DAG and has three possible variations or heuristics that can be chosen optionally: **Non-recursive** `FIND` function, **Heuristic** `UNION`, **Forbidden Set** use.

This project is written in **Java** language, used due to many build-in optimized datas stucture implemented and easy testing enviroment.

Along the solver, there is a **generator** for the creation of a synthetic set of literals customizable by the user (under a cert extent), and a **parser** for a limited set of **QF_UF SMT-LIB** files.

2 Project Folder

The project folder is organized in the following way:

- **analyzed**: this folder contains output files used in table and graphic.
- **generator**: this folder contains the properties files used by `Generator`
- **input**: this folder contains the txt files that have the formula or set of literals to be checked, those files use the basic syntax of the solver.
- **output**: this folder contains the result submitted in the input folder.
- **smtlib_input**: this folder contains the .smt2 files from SMT-LIB without edit.

The other folders containain java files and compiled class files.

The main class is `Congruence Closure Solver` which acts as **central manager** and is responsible for reading the name file given in input (that has to be in `input`, `generator`, `smtlib_input` folder based on the type of file), the options and retrieves the formula using a `FormulaReader`. The formula will be passed to a **parser** and then to the **algorithm**.

3 Input

The program accepts both a **set of literals** delimited by `;` and a formula with **logical connectives**.

It is also possible to submit a generator file with `.properties` extension to create a customizable set of literals, with **random** mechanisms.

Or to submit directly a `.smt2` file from QF_UF SMT-LIB, not all the files are compatible since the SMT-LIB language is very complex.

More info about the syntax is available in the `README` file.

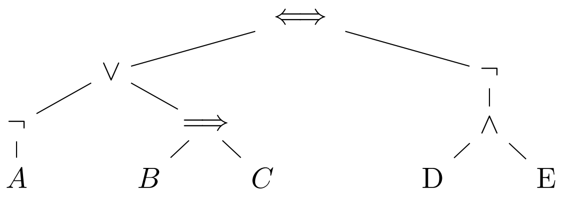
4 Parsing

The retrieved formula is submitted by a chain of parsers:

1. `DNFParser`: after dropping the **existential** quantifiers and the **universal** quantifiers, parse the input formula and cast it in a `DNFTree`, a n-tree where the nodes are **logical connectives** or **predicates**, and the edge are the scope of the logical connectives.

   (More info about the precedence used in in the `README` file)

   For example, the formula $[\neg A \vee [B \implies C]] \iff [\neg[D \wedge E]]$ is encoded as:

2. `LogicParser`: use the `DNFTrasformer` handle a chain of transformation that reduces the `DNFTree` to a tree with an `OR` root that has only have `AND` children, which will have only predicate children (already negated if is the case), which will be the **leaves**.

   Then cast the leaves into an `ArrayList<String>` which will represent the list of **cubes** to solve one at a time.

   Another task that the `LogicParser` performs is the **splitting** required upon receiving a formula that contains an instance of `select(score(...))`, the parser checks the occurrence and splits the formula in the two versions since it has handled a list of cubes (`OR`) than the parser removed the split cube, and enqueue the new ones, ready to be checked at the end, and performing the operation again if necessary.

3. `Term Parser`: it receives a set of clauses one per time, and converts the literals in **nodes**.

   It handles both predicate $P(a)$ or $\neg P(a)$ and converts them inequalities using a true constant, that the user cannot insert in the input: `P(a) = °` or `P(a) != °`.

   Then it splits each (dis)equality and makes a `Node` object for each of them with an associated (`int`) `id`, each node is contained in an `HashMap<Integer,Node>` for easy access, while the pair of id of term `L` associated with a (dis)equality are stored in two `ArrayList<Integer[]>`, in this case `L[0]` is the left term while `L[1]` is the right term.

   The parser also updates the **Forbidden Sets** of the nodes even if they will not be used if the option is not toggled.

   While processing the terms the parser recognizes the presence of a reserved name, if the given formula is in one of the three compatible **theories**. Since the parser can not handle a combination of theories, if two reserved names from the theory of **list** and **array** are detected, then the parser will consider the formula in the theory of **equality**, using the reserved name function as free symbols.

   The last task is address to do, is the **transformation** required by the list of theory, such `atom(a)` into `cons(a_1,a_2) = a`, applying **projection atoms** and adding disequalities such as: `cons(x,y) != atom(z)` for each `x,y,z` occurrence.

5 Congruence Closure Algorithm

After all parsing steps, for each cube, the `TermParser` forward the map of `Node` objects and the `ArrayList<Integer[]>` of (dis)equalities to the `CongruenceClosureAlgorithm` implemented with a `CongruenceClosureDAG`; this data structure along with `Node` are implemented following the definition in Bradley, Manna, et al. [1].

The main task of `CongruenceClosureAlgorithm` is to instruct `CongruenceClosureDAG` to `MERGE` all the nodes in the equalities list, once there are no more nodes to merge, checks that the pair of id in the disequalities list is not in the same **ccpar set (implemented as a `HashSet<Integer>`)**, and return a `bool` that indicates the satisfiability.

If the **Forbidden Set** option is enabled, then the `CongruenceClosureDAG` will prevent the `MERGE` of two representatives of disequalities, and return to the algorithm the conflict, which will return unsat. If no conflicts are found, then `CongruenceClosureAlgorithm` will skip the checking disequalities part.

If the **Heuristic Union** or the **Recursive Find** options are enabled then the `EUR_UNION` and `REC_FIND` version will be used as described in Detlefs, et al. [2].

6 Output

The result will be written in a txt file in the output with the same name as the file in the input one. If the **verbose** `"v"` option is enabled, then all the steps of the `CongruenceClosureDAG` will be written, it is advisable to not enable this option is for long formula or in CNF since the transformation in DNF is very prolific.

At the end of the file will be written the result and the time elapsed for the solving, which do not take into account the time in DNF transformation.

7 Sintetic Generator

Along the solver is present a `Generator` which will write a set of literals based on the user's **choices**.

The user can choose the **signature** (constant, functions and predicate), the max **arity** for functions and predicates, the max **depth** of a function or predicate, the number of **cubes** to generate, the **probabilities** of the presence of constants, function or predicates, and the optionally the **seed**, so that the randomly generated formula can be replicated, with same parameters.

It is advised to restrain depth and artity to 10, and cubes to 100, since the Java memory cannot handle too many `Node` structures.

The generator will place the synthetic list of literals in a txt file in the input folder with the same name as the properties file, concatenated with the seed.

The execution of the solver is automatic after the submission of the file name.

8 SMT-LIB

Since the **SMT-LIB** language features a lot of definition and syntax, only a restricted subset of formulas of the QF_UF have been made compatible with the `SMTLIBParser` in particular the formulas present in `eq_diamond` from SMT Workshop 2005.

In these files are present only an `assert` check for a formula written in this form:

$$\bigwedge_i \left(\bigvee_j x_{i,j} \simeq y_{i,j} \wedge z_{i,j} \simeq w_{i,j} \right) \wedge u \not\simeq v \quad u, v \in \{x_{i,j}, y_{i,j}, z_{i,j}, w_{i,j} \mid \forall i, j\} \tag{1}$$

The **length** of the formula is determined by the number of equations, (eq_diamond4 is larger than eq_diamond2) and the result of all formulas are UNSAT, so are been used to test the heuristics implemented in this project.

As for the `Generator`, the smt2 files have to be placed in the **smtlib_input** and an equivalent parsed formula with the project syntex, will be written in a txt file in the input folder with the same name.

The execution of the solver is automatic after the submission of the file name.

Since the DNF transformation is not optimized, it is not advisable to run eq_diamond15 or larger.

9 Performance Analysis

The solver works with all the files located in `input` folder, but some of them have been analyzed using different option, these results can be found in `analyzed` folder and used to create the following table and graphic.

Name	**Recursive**	**H-Union**	**Forbidden**	**Time**	**Result**
ArrayBM9.8d_r	X			0.000s	UNSAT
ArrayBM9.8d_re	X	X		0.001s	UNSAT
ArrayBM9.8d_rf	X		X	0.000s	UNSAT
EqMB1.8.txt				0.002s	SAT
EqMB1.8_r.txt	X			0.003s	SAT
EqMB1.8_e.txt		X		0.002s	SAT
EqMB1.8_f.txt			X	0.002s	SAT
ListBM9.6b				0.011s	UNSAT
ListBM9.6b_ref.txt	X	X	X	0.009s	UNSAT
eq_diamond10_r	X			0.613s	UNSAT
eq_diamond10_ef		X	X	0.316s	UNSAT
eq_diamond12_r	X			6.246s	UNSAT
eq_diamond12_ef		X	X	6.143s	UNSAT
eq_diamond14_r.smt2	X			91.30s	UNSAT
eq_diamond14_ef.smt2		X	X	89.49s	UNSAT
eq_diamond14.smt2				88.06s	UNSAT
eq_diamond14_f.smt2			X	87.10s	UNSAT
GEN100.properties				0.074s	SAT
GEN100_r.properties	X			0.067s	SAT
GEN100_e.properties		X		0.076s	SAT
GEN100_f.properties			X	0.080s	SAT
GEN100_re.properties	X	X		0.071s	SAT
GEN100_rf.properties	X		X	0.070s	SAT
GEN100_ef.properties		X	X	0.077s	SAT
GEN100_ref.properties	X	X	X	0.072s	SAT

Table 1: Test Results Table

For short formulas, the different options have no significant impact.

However, for medium-sized formulas, the recursive version of `FIND` appears to boost performance. The idea behind the **Non-Recursive** approach is to reduce the recursive calls of `FIND` at the cost of updating the representatives of all nodes associated with a representative node that is not chosen during the `MERGE`. This will be referred to as the **Non-Recursive Burden**.

For large formulas, the **Forbidden Set** significantly improves the speed of detecting conflicts in most of the cases. However, using **Heuristic Union** slows down the process. This could be due to the implementation of the heuristic, as the representative node is chosen based on the larger ccpar set. Each time there is a `MERGE`, there may be a potential sub-`MERGE` between different but congruent ccpar elements of the two terms. If both terms have large ccpar sets, this increases the resolution time. This phenomenon will be referred to as the **H-Union Burden**.

The elapsed times vary depending on whether only recursive `FIND` (without heuristics) is used or all heuristics are applied, based on the formula size. The following graphic illustrates which formula sizes are better served by these approaches.

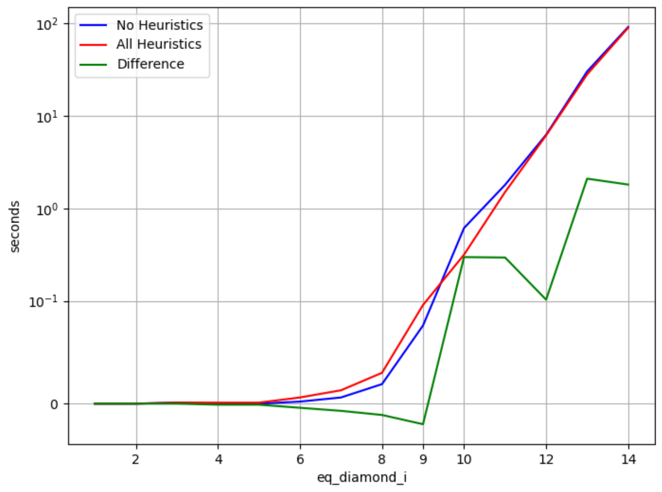


Figure 1: Comparison of Performance With and Without Heuristics

As the graphic shows, the heuristics become helpful after `eq_diamond9.smt2`, as indicated by the green line (Heuristic Time - No Heuristic Time).

Between `eq_diamond5.smt2` and `eq_diamond9.smt2`, the **Non-Recursive Burden** and **H-Union Burden** are heavier.

The algorithm spends significant time updating the representatives, which happens frequently due to the heuristics. This is not justifiable since the formulas are not large. A more distributed ccpar set size, without requiring checks from all nodes for representative changes, could be beneficial.

Meanwhile, the **Forbidden Set** does not seem to help much, as passing all the IDs in the set between representatives and non-representatives can be expensive. Another reason could be that the formula size is insufficient to justify this overhead.

For `eq_diamond9.smt2` and larger formulas, the difference between using heuristics and not is noticeable. With many literals to check, recursion and distribution of large ccpar sizes can be more expensive than the **Non-Recursive Burden** and **H-Union Burden**.

References

[1] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Berlin, Heidelberg, Berlin, Heidelberg, 1 edition, 2007. eBook ISBN: 978-3-540-74113-8.

[2] David Detlefs, Greg Nelson, and James Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52, 09 2003.