Planning and Automated Reasoning - Automated Reasoning

Project: Implementation of the congruence closure algorithm

Cristian Morasso - VR492389

June 23, 2023

1 Scope

The purpose of this project is to implement the congruence closure algorithm using a DAG to determine the satisfiability of a set of equalities and inequalities in the quantifier free fragment of the theory of equality.

2 Implementation

2.1 Main contributes

- Congruence Closure (Node Class and Solve function)
- SMT parser
- Formula parser

2.2 Congruence Closer Class

First of all, to create a DAG, we must define a **Node** class as follows.:

Listing 1: Node DAG class

```
1    class node_DAG:
2        def __init__(self, id, fn, string, args = []):
3            self.id = id  #id integer
4            self.fn = fn  #function symbol
5            self.find = id #rapresentative id
6            self.args = args #list of id (argoments)
7            self.ccpar = set() #set id (parents)
8            self.string = string # string rappresentation of a node
```

This class represents the main structure of the algorithm, which we use it as a data structure. We added a new attribute *string* which represents the string version of that node, we don't use a *__str__* because we would need to access to the graph and know how to represent the args, so we prefer to save the string as an attribute, furthermore we will use this camp to search if the node's already being created.

Then there is the **CC_DAG** class, which contains all methods to deal with the DAG, in particular there are the algorithm functions:

- *Union*
- *Congruent*
- *Merge*
- *Find*
- *NODE*

- *CCPAR*

and then the **solve** function to determine, after the process, if the set is *SAT* or *UNSAT*. Note that at line 3 is implemented the forbidden list technique, which aims to speed up the process.I implemented it here because the book said so, however implementing this in an earlier stage of the code would have been better for speed performances.

Listing 2: solve with forbidden list implementation

```
1    def solve(self):
2        for eq in self.equalities:
3            if eq in self.inequalities or eq[::-1] in
                 self.inequalities: return 1,ineq
4            self.merge(eq[0],eq[1])
5        for ineq in self.inequalities:
6            find1,find2 = self.find(ineq[0]),self.find(ineq[1])
7            if find1 == find2:
8                return 1,ineq # UNSAT if the 2 finds of an ineq are
                     same
9        return 0, ""
```

2.2.1 *Union*

Note I updated the original union method to implement *a non arbitrary choice of the representative of new class* by picking the representative the bigger, I also remove if the first id is already in the other ccpar, in this way:

Listing 3: Union code with priority

```
1    def union(self,id1,id2):
2        n1 = self.NODE(self.find(id1))
3        n2 = self.NODE(self.find(id2))
4        if len(n1.ccpar) >len(n2.ccpar):
5            n2.find = copy.copy(self.find(id1))
6            n1.ccpar.update(n2.ccpar)
7            n2.ccpar = set()
8            if n1.id in n1.ccpar:
9                n1.ccpar.remove(n1.id)
10       else:
11           n1.find = copy.copy(self.find(id2))
12           n2.ccpar.update(n1.ccpar)
13           n1.ccpar = set()
14           if n2.id in n2.ccpar:
15               n2.ccpar.remove(n2.id)
```

2.2.2 `fix find`

I added this method which is called after every unions in the merge method, it aims to avoid chain links for the find attribute. This correction is helpful for the trace when we draw the dotted lines (find links).

In the end there are some plotting functions that allow to plot the DAG before the processing and after with the dotted arrow.z

2.3 STM parser

The idea of this file is to provide a parser which is able to read a SMT file returning a list of AND clauses (2.3.1). And then the other method (2.3.2) parses this list element by element in order to extract from the element the formulas and the atoms.

In particular:

2.3.1 `parse`

This method gets as input the file name, it reads the file and returns the formulas split on the OR, (this software can only parse function without OR or with the OR at the higher level (DNF form) better specified at 3.1). This method also read from the SMT file the ground truth of the problem, if not we return *UNKNOWN*.

2.3.2 `parse_and_clause`

Firstly we split the and clause on the *AND* in order to get the single formula, then we extract the atom from each formula, which will be used to create the nodes of the DAG.

2.4 Formula parser

In this file we define a *parse_equations* 2.4.3 and the class *parse_atoms* with its 2 functions.

2.4.1 `parse`

For each atom, it calls the rec build function to process all atoms if it's is a new atom.

2.4.2 `rec build`

This is the main function that parses all the formulas going deep in the arguments in a recursively way. When we reach the inner argument we check if its node is already in the dictionary, if not we create it with an incremental id and add it to the dict. After that we return to the previous

recursive call and add the nodes to the arguments of the current atom that we are processing; we do this till we finish all the atoms. Doing this we create a node giving *node.id*, *node.fn*, *node.string* and *node.args*

2.4.3 `parse_equations`

This function processes all equations and splits them in equations and inequalities, saving it as a pair of atoms.

3 How to run the code

The code is deploy as a colab notebook, to run it you have to press Runtime and run all the cells (Ctrl+F9). The code description is on the notebook as markdown cells.

3.1 How to input

This software accepts only SMT files, because SMT is the most common format for this type of problem. The SMT files is in CNF form which is not good for us, because the Congruence Closure Algorithm works only on *AND* clauses, therefore we have opted to accept input without OR or already in DNF. For DNF case the algorithm splits the entire input into and clauses on the *ORs*. To try your input you can load the SMT file on the workspace (you can easily drag and drop the file or the directory), then change the path in the code and run the notebook, (if is a directory i suggest to run it without verbose flag).

Listing 4: Example of the SMT file

```
1 (set-info :smt-lib-version 2.6)
2 (set-logic QF_UF)
3 (set-info :source |
4 Source: The calculus of computation (Bradley-Manna) |)
5 (set-info :category "crafted")
6 (set-info :status unsat)
7 (declare-sort S1 0)
8 (declare-fun f (S1) S1)
9 (declare-fun a () S1)
10 (assert (let ((t1 (f a)) (t2 (f t1 )) (t3 (f t2 )) (t4 (f t3 )) (t5
       (f t4 ))) (and (= t3 a) (= t5 a) (not (= t1 a)) ) ))
11 (check-sat)
12 (exit)
```

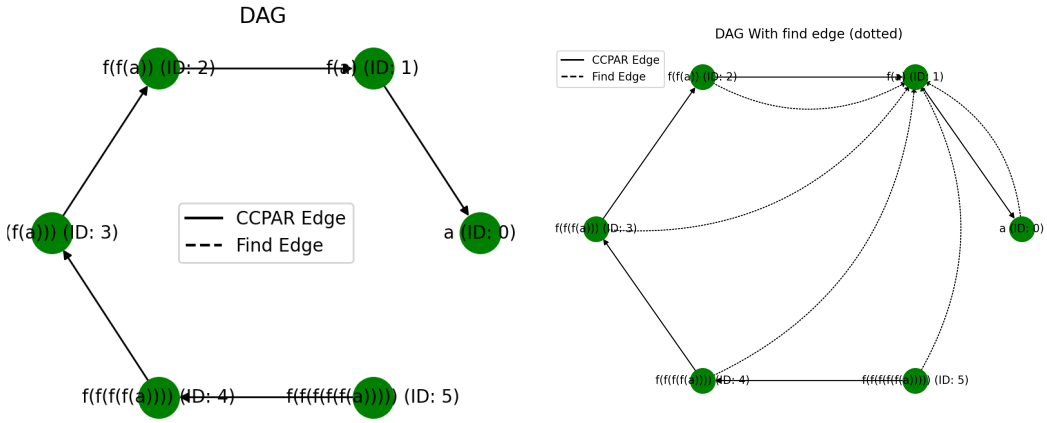
4 Results

Here I will show an example of the result (the output of listing 4). All the results are available when you run the code on Google Colab.

Listing 5: Result example

```
Formulas: ['f(f(f(a))) = a', 'f(f(f(f(f(a))))) = a', '! (f(a) = a)']
Solver output: UNSAT
Ground Truth: UNSAT
Solved in: 10.0 ms
```

These two DAGs below 4 represent the DAGs before and after the process, in the second one we can see the dotted edges which represent the *Find links* for each atom.



(a) DAG before the process, only the CCPAR edges

(b) DAG processed with representative edges

Figure 1: DAGs in a graphic representation

5 Assumptions & Problems

The main assumption is the constraint on the input format. The main problem was about the low number of inputs that the software was able to process so we needed to implement a SMT parser to work with a higher number of inputs, despite that we found another problem, SMT files are written in CNF form but the software works with DNF, so I decided to not dive deep into this problem.