Congruence Closure Algorithm

Leonardo Zecchin VR462541

June 19, 2023

1 Introduction

This paper presents the research conducted within the scope of the Automatic Reasoning course for the academic year 2022/2023. Specifically, the focus lies on the implementation of the Congruence Closure Algorithm utilizing Directed Acyclic Graphs (DAGs).
The content of the project can be found at the following link: Leonardo Zecchin's project.

2 Implementation

2.1 Project structure

1. The *mainProgram* program implements the algorithm by reading an input file that contains formulas in normal form, e.g., `f(a,b)=a and f(f(a,b),b)!=a`.

2. The *theParser.py* program implements the algorithm by reading an input file that contains the formulas that need to be brought into DNF form and then brought to normal form, e.g., `and(eq(f(a,b),a),dis(f(f(a,b),b),a))` becomes `f(a,b)=a and f(f(a,b),b)!=a`;
or `imply(eq(x,g(y,z)),eq(f(x),f(g(y,z))))` becomes `and(eq(x,g(y,z)),dis(f(x),f(g(y,z))))` `and after x=g(y,z) and f(x)!=f(g(y,z))`.

Within the `code` folder are the codes that are used by the main programs, in particular `cca` is used for the **Congruence Closure Algorithm**, while the other programs were used during implementation.

Inside the `classes` folder, you will find two important classes: `dag` and `node`, which are utilized by the algorithm.

In the `input` and `output` folders, there are two types of files:

1. `input.txt` and `output.txt`: the former contains the formulas in the normal form, and in the latter, you will find the algorithm's resulting outcomes.

2. `inputToParser.txt` and `outputToParser.txt`: the former contains formulas that must be parsed, and in the latter, you will find the algorithm's resulting outcomes.

2.2 The Algorithm

The structure of the algorithm is the following:
Given Σ_{E}-formula

$$F : s_1 = t_1 \wedge \cdots \wedge s_m = t_m \wedge s_{m+1} \neq t_{m+1} \wedge \cdots \wedge s_n \neq t_n$$

with subterm set S_F, perform the following steps:

1. Construct the initial DAG for the subterm set S_F.

2. For $i \in \{1, \ldots, m\}$, MERGE $s_i t_i$.

3. If FIND s_i = FIND t_i for some $i \in \{m+1, \ldots, n\}$, return unsatisfiable.

4. Otherwise (if FIND $s_i \neq$ FIND t_i for all $i \in \{m+1, \ldots, n\}$) return satisfiable.

Inside the `dag.py` you will find the implementation of the algorithm's functions: *MERGE*, *UNION*, *CONGRUENT*, *CCPAR*, *FIND* and *NODE*, the specific explanation of functions is outside the scope of the paper.

The implementation of the algorithm is within the **cca.py** in particular the function is the `congruenceClosureAlgorith` which takes in input:

1. F_plus: it contains the formulas with equality (=);
2. F_minus: it contains the formulas with disequality ($\neq$);
3. Sf: the subterm set;
4. new_dag: is the DAG that represents the subterm set.

```
def congruenceClosureAlgorithm(F_plus, F_minus,Sf,new_dag):
    for f in F_plus:
        #Step 1
        idx1,idx2 = getIndex(f,Sf)
        new_dag.MERGE(idx1,idx2)
    #Step 2
    for f in F_minus:
        idx1,idx2 = getIndex(f,Sf)
        if new_dag.FIND(idx1) == new_dag.FIND(idx2):
            return False
        else:
            return True
```

In the `congruenceClosureAlgorithm` there are the implementation the second, the third and fourth steps.

2.3 Differences with pseudocode

The only difference between my code and the pseudocode is in the `UNION` function:

```
def UNION(self, id1: int, id2: int) -> None:
    print(f"UNION {id1} {id2}")
    n1 = self.NODE(id1)
    n2 = self.NODE(id2)
    n1_ccpar = self.CCPAR(id1)
    n2_ccpar = self.CCPAR(id2)
    if self.FIND(n1.find) != n1.id:
        self.NODE(self.FIND(n1.find)).find = n2.find
    else:
        n1.find = n2.find
    n2.ccpar = n2_ccpar+ n1_ccpar
    n1.ccpar = []
```

I introduced the `if/else` statement to address scenarios in which the algorithm needs to modify the find field of the n1 node, but it is connected to another node. In such cases, I must also modify the find field of the node linked to n1. Additionally, a notable distinction is that UNION stores the respective **CCPAR** values in `n1_ccpar` and `n2_ccpar` prior to altering these fields.

3 Results

In this section we show the results obtain with the algorithm.

3.1 Table

Experiments			
Formulas	DNF	Satisfiability	Time execution
-imply(eq(x,g(y,z)),eq(f(x),f(g(y,z))))	x=g(y,z)andf(x)!=f(g(y,z))	UNSAT	0.0004217
-and(eq(f(a,b),a),dis(f(f(a,b),b),a))	f(a,b)=aandf(f(a,b),b)!=a	UNSAT	0.000449
-and(eq(f(f(f(a))),a),and (eq(f(f(f(f(f(a))))),a),dis(f(a),a)))	f(f(f(a)))=aand f(f(f(f(f(a)))))=a and f(a)!=a	UNSAT	0.0006108
-and(eq(f(f(f(a))),f(a)), and(eq(f(f(a)),a),dis(f(a),a)))	f(f(f(a)))=f(a)and f(f(a))=a andf(a)!=a	SAT	0.000388
-and(eq(x,g(x,z)),dis(f(x),f(g(y,z))))	x=g(x,z)and f(x)!=f(g(y,z))	SAT	0.00032