**PLANNING AND AUTOMATED REASONING**

**Automated Reasoning, Academic Year 2022-23**

Student: Ghezzo Francesco VR496402

**Project: Implementation of the congruence closure algorithm**

**INTRODUCTION**

In this project I implement the congruence closure algorithm with DAG for the satisfiability of a set of equalities and disequalities in the quantifier-free fragment of the theory of equality. The algorithm was explained in class and is described in Sect. 9.3 of the Bradley-Manna textbook.

The congruence closure algorithm is a technique used in automated theorem proving and equivalence checking. Here's a brief description:  
The congruence closure algorithm maintains an equivalence relation over a set of elements, represented as a partition.

1. Initially, each element forms its own equivalence class.
2. The algorithm processes congruence rules, which state that if two expressions are equal, then their corresponding elements are in the same equivalence class.
3. It iteratively applies these congruence rules to merge equivalence classes.
4. During each iteration, it checks pairs of elements in different classes for congruence.
5. If two elements are congruent, it merges their equivalence classes into a single class.
6. The algorithm also maintains a "merge find" data structure for efficient equivalence class manipulation.
7. It terminates when no further merges can be made.
8. The resulting equivalence classes represent the congruence closure of the initial set of elements.
9. Check using the solver() implemented if the input is SAT/UNSAT.

**BASIC OPERATIONS (Bradley-Manna)**

* NODE(self, node\_id:int)
* FIND(self, node\_id:int)
* UNION(self, n1:int, n2:int) (non-arbitrary choice)

def UNION(self, n1:int, n2:int):

        n1 = self.NODE(self.FIND(n1))

        n2 = self.NODE(self.FIND(n2))

        if len(n1.ccpar) < len(n2.ccpar):

            n1.find = n2.find

            n2.ccpar = n1.ccpar.union(n2.ccpar)

            n1.ccpar = set()

        else:

            n2.find = n1.find

            n1.ccpar = n2.ccpar.union(n1.ccpar)

            n2.ccpar = set()

* UNION2(self, n1:int, n2:int) (not used)
* CCPAR(self, node\_id:int)
* CONGRUENT(self, node\_id1:int, node\_id2:int)
* MERGE(self, node\_id1:int, node\_id2:int, count: int)

**FUNCTIONS**

**Solver.py file:**

class Node:

    def \_\_init\_\_(self, id:int, fn:str, args:list, find:int, ccpar:set):

        self.id = id

        self.fn = fn

        self.args = args

        self.find = find

        self.ccpar = ccpar

    def \_\_hash\_\_(self):

        return hash(self.fn) \* hash(tuple(self.args))

    def \_\_eq\_\_(self, other):

        return hash(self) == hash(other)

class DAG:

* Add\_father(self, id)
* Print\_node(self, node\_id:int)
* Print\_nodes(self)
* Node\_string(self, id)
* Complete\_ccpar self
* Add\_forbidden\_list(self, forbidden\_list:set)
* Add\_equalities(self, equalities:list)
* Add\_inequalities(self, inequalities:list)
* Remove\_node(self, node:Node)
* Add\_node(self, node:Node)
* Solve(self) (with forbidden list)

    def solve(self):

        count = 0

        for eq in self.equalities:

            val1, val2 = eq[0], eq[1]

            if (val1, val2) in self.forbidden\_list: return "UNSAT -> forbidden list", count

            if (val2, val1) in self.forbidden\_list: return "UNSAT -> forbidden list", count

            count = self.MERGE(eq[0], eq[1], count)

        for ineq in self.inequalities:

            val1, val2 = self.FIND(ineq[0]), self.FIND(ineq[1])

            if val1 == val2:

                return "UNSAT", count

        return "SAT", count

* Visualize\_dag(dag)
* Print\_final\_graph(dag)

**Custom\_parser.py file:**

class Parser:

    def \_\_init\_\_(self, graph):

        self.customParser = nestedExpr('(', ')')

        self.graph = graph

        self.ids = set()

        self.atoms\_dict = dict()

* Parse(self, input)
* Parse\_clause(self, atom\_as\_list:list, unique\_atoms)
* NewId(self)

Out of Parser class but inside custom\_parser.py:

* split\_string\_by\_or(input\_string)
* eq\_ineq(equations, atoms\_dict)
* my\_alg\_OR(clause, list\_sat)
* my\_alg\_AND(line)
* my\_alg\_SMT(filename, parser)

**ALGORITHM**

I defined a graph-based data structure for representing all members of the subterm set SF of a ΣE-formula F. Each node represents a subterm. Congruence classes are stored within this data structure via references between nodes.

The first problem I encountered was to parse the input (.txt or .smt2) to be used inside the algorithm.

I created two functions: parse() and parse\_clause() to recursive parse the input from .txt file (smt\_parser() and parse() for .smt2 files) and transform it in a way I can manipulate it to create the graphs, starting from:

1. recursively enter the most nested costant;
2. saving the constant and creating the relative node;
3. recursively going out saving the functions/constant as new node controlling to not create the same node two times;
4. when I create the nodes, I set the “ID” (unique), the “fn” (as the Bradley-Manna does), the “find” (same as the ID) and the “args” (children of the node);
5. once the graph is created, I call the function complete\_ccpar() to fill the “ccpar” field;
6. then I fill the inequalities list, the equalities list and the forbidden list;
7. I call the solve() function with the forbidden list implementation (also with the reverse forbidden list). The solve function is the same as the Bradley-Manna implementation but with forbidden list added.
8. In the end, I print the results:
   1. Set of clauses (formula) used in this iteration;
   2. SAT/UNSAT;
   3. Number of merge;
   4. Time spent for the single set of clauses.

**INPUT**

The inputs can be given in two formats:

1. as a .txt file, where each line corresponds to a set of clauses (formula), given in CNF;
2. as a .smt2 file, where where each file corresponds to a set of clauses (formula), given in smt2 format and translate by the function smt\_parser in CNF.

Both inputs can be given with AND and OR or both together. The unique problem is when the OR is nested inside function of functions in the smt2 files, this doesn’t work in my implementation.

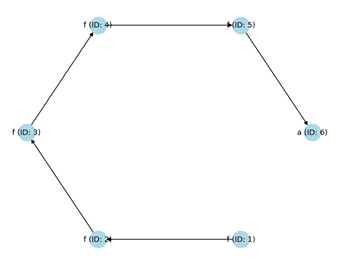
To run my implementation, there is a flag at the start of the main that establish which input you want to use, if a .txt or a .smt2. This flag has to be setted True if the user want to pass a .txt file, False if the file has .smt2 extention.

By default, none graphs will be give as output (because it slows a lot the execution). To be seen, it must decomment the line “solver.visualize\_dag()”.

**EXAMPLE**

Function*: f(f(f(a))) = a & f(f(f(f(f(a))))) = a & f(a) != a*

Related graph:



Related output:

A picture containing text, screenshot, font, software

Description automatically generated

**RESULTS**

.txt results:

|  |  |  |  |
| --- | --- | --- | --- |
| **LINES (with & and or)** | **SAT/UNSAT** | **# MERGES** | **TIME (ms)** |
| 1 & | UNSAT | 2 | 7.3 |
| 2 &, or | UNSAT | 3 | 6.6 |
| 3 & | SAT | 0 | 3 |
| 4 & | UNSAT | 1 | 4.2 |
| 5 &, or | UNSAT | 3 | 7.1 |
| 6 &, or | SAT | 0 | 3.7 |
| 7 & | UNSAT -> forbidden list | 0 | 1.8 |

Every time in the table is a mean over three run.

.smt2 results:

|  |  |  |  |
| --- | --- | --- | --- |
| **#file (with & and or)** | **SAT/UNSAT** | **# MERGES** | **TIME (ms)** |
| 1 & | UNSAT | 1 | 7.1 |
| 2 & | UNSAT | 2 | 9 |
| 3 & | UNSAT | 1 | 8.6 |
| 4 & | UNSAT | 1 | 10 |
| 5 & | UNSAT | 2 | 14 |
| 6 & | UNSAT | 1 | 8.6 |
| 7 & | SAT | 1 | 6.4 |
| 8 & | UNSAT | 1 | 9 |
| 9 & | UNSAT | 1 | 11 |
| 10 &, or | UNSAT | 0 | 7.1 |
| 11 & | UNSAT | 3 | 12 |
| 12 & | UNSAT | 1 | 11 |
| 13 & | UNSAT -> forbidden list | 0 | 6 |

Every time in the table is a mean over three run.

**COMMENTS/ANALISYS/OPTIMIZATIONS**

I implement three type of optimizations:

1. The forbidden\_list: I check if there is at least one equalities inside the forbidden list, which is a list of inequalities. I implement also a reverse check because, to use an example:  
   *“a=b & b!=a”* return “UNSAT” from a normal forbidden\_list, but doing also the reverse check, it will return “UNSAT -> forbidden list”. This control speeds up execution and run time.
2. Union() with non-arbitrary choice: I check which node during the union has more parents than the other. I do the union on this parent.
3. When is running the SAT-solver ( solver() ) over a formula with “or” inside, I don’t let the algorithm to run to the end if it encounters any “SAT” as a return. Because it needs only one SAT over the entire formula.  
   example: “*f(a) = f(a) or f(f(f(a))) = a & f(f(f(f(f(a))))) = a & f(a) != a”*   
   Since the first clause “f(a) = f(a)” is SAT, the algorithm doesn’t check the other clauses but then it return directly “SAT”.

I did some analysis as displayed in the tables before, I didn’t manage to properly parse the bigger files inside the repository (smt2 lib) because they have a lot of nested OR and I should transform completely the CNF into DNF.   
This causes the execution of the algorithm to become infeasible in my local machine.

To be mentioned, if the input is in format of “ ”Extending Sledgehammer with SMT Solvers" by Jasmin Blanchette, Sascha Bohme, and Lawrence C. Paulson, CADE 2011. Translated to SMT2 by Andrew Reynolds and Morgan Deters”, my parser doesn’t work properly because the rappresentation of nested function has a small difference.

**BIBLIOGRAPHY**

A. R. Bradley and Z. Manna, The calculus of computation: *decision procedures with applications to*

*verification*. Springer Science & Business Media, 2007.

JasminBlanchette, Sascha Bohme, and Lawrence C. Paulson, CADE *“Extending Sledgehammer with SMT*

*Solvers”,* 2011. Translated to SMT2 by Andrew Reynolds and Morgan Deters