Planning and Automated Reasoning
– AUTOMATED REASONING-
II term, Academic Year 2022-23

Project:
Implementation of the congruence closure algorithm

https://github.com/SerenaDeAntoni/Congruence_Closure_Algorithm.git

Author:
Serena De Antoni
VR487515

1. INTRODUCTION

In this project I implement the congruence closure algorithm with DAG for the satisfiability of a set of equalities and inequalities in the quantifier-free fragment of the theory of equality.
The algorithm implement was explained in class and is described in Sect. 9.3 of the Bradley-Manna textbook, and I considered the variant of the forbidden list - when calling MERGE s t if s is in the forbidden list of t or vice versa, return 'unsat' – and of the non-arbitrary choice of the representative of the new class in the UNION function - pick the one with the largest ccpar set.
I chose Python as programming language and to test my program I considered sets of equalities and inequalities from the Bradley-Manna textbook and some benchmarks with ".smt2" extension.
My code is in the following link and to be able to run it follow the instructions at the colab link in the README.md file: https://github.com/SerenaDeAntoni/Congruence_Closure_Algorithm.git

2. CONGRUENCE CLOSURE ALGORITHM

2.1 Nodes

In the congruent closure algorithm, the objective is to establish the relationship of equivalence among the different parts of a formula, or to demonstrate that such a relationship does not exist. It uses a Directed Acyclic Graph (DAG) to represent terms of the subterm set.
In the CC Algorithm, the terms used to represent nodes have the following components:s

1. id: It is usually a number and it refers to the unique identifier assigned to each node. It distinguishes one node from another and helps identify and keep track of different nodes.
2. fn (Function Symbol): It is usually a string and it indicates the operation or relation performed by the term.
3. args (Arguments): It is usually a id list and it indicates the child nodes. These child nodes are the sub-terms on which the main operation or relationship (represented by the function symbol) operates
4. find: It is a way to find the main representative element for a group of nodes that are equivalent.
5. ccpar (Congruence Closure Parent): If a node is the representative for its congruence class, then its ccpar field stores the set of all parents of all nodes in its congruence class. A non-representative node's ccpar field is empty.

```
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
```

2.2 Basic Operations

The main defined functions for the Congruent Closure Algorithm are:

- NODE: NODE *i* returns the node n with id

```
    def NODE(self, node_id:int):
        for node in self.nodes:
            if node.id == node_id:
                return node
        print('Node not found')
```

- FIND: The find function returns the representative of a node's equivalence class. It follows find edges until it finds a self-loop.

```
    def FIND(self, node_id:int):
        node = self.NODE(node_id)
        if node.find == node_id: return node_id
        return self.FIND(node.find)
```

- UNION: The union function returns the union of two equivalence classes, given two node identities i1 and i2. In this case I implement Non-arbitrary choice of the representative of the new class in the UNION function, so I pick the one with the largest ccpar set.

```
    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()
```

- CCPAR: The function ccpar *i* returns the parents of all nodes in i's congruence class:

```
def CCPAR(self, node_id:int):
        return self.NODE(self.FIND(node_id)).ccpar
```

- CONGRUENT: congruent *i1 i2* tests whether *i1* and *i2* are congruent.

```
    def CONGRUENT(self, node_id1:int, node_id2:int):
        n1 = self.NODE(node_id1)
        n2 = self.NODE(node_id2)
        res = True if (n1.fn == n2.fn and len(n1.args) ==
len(n2.args) and [self.FIND(n1.args[i]) == self.FIND(n2.args[i])
for i in range(len(n1.args))]) else False return res
```

- MERGE: merge *i1 i2* merges the congruence classes of *i1* and *i2*

```
    def MERGE(self, node_id1:int, node_id2:int, count: int):
        if self.FIND(node_id1) != self.FIND(node_id2):
            p1 = self.CCPAR(node_id1)
            p2 = self.CCPAR(node_id2)
            self.UNION(node_id1, node_id2)
            for t1, t2 in list(itertools.product(p1, p2)):
                if self.FIND(t1) != self.FIND(t2) and
self.CONGRUENT(t1, t1):
                    self.MERGE(t1, t2, count)
                    return count +1
                else:
                    return count
        return count
```

2.3 Other functions

There are other functions such as:

- `def add_forbidden_list(self, forbidden_list:set):` It creates the forbidden list
- `def add_eq(self, equalities:list):` it creates the equality list
- `def add_ineq(self, inequalities:list):` it creates the inequality list
- `def add_node(self, node:Node):` Appends a Node object to the nodes attribute.
- `def complete_ccpar(self):` Adds fathers to each node in the nodes attribute. It iterates over each node and calls the add_father function for that node's ID.
- `def add_father(self, id):` Adds the current node ID as a father to each node in the args attribute of the node with the given ID.
- `def print_node(self, node_id:int):` Prints the details of the Node object with the given node_id, including its ID, fn, args, find, and ccpar attributes.
- `def print_nodes(self):` Prints the details of each node in the nodes attribute by calling the print_node function for each node.
- `def node_string(self, id):` Generates a string representation of the Node object with the given id. If the node has arguments, it recursively generates string representations for each argument and combines them with the node's function name.
- `def visualize_dag(dag):` Visualizes the directed acyclic graph (DAG)

In the Congruent_Closure_Algorithm_with_DAG class there is also the solve() function in which I control if my pair in the equality list are in the forbidden list, and if it is I return "UNSAT -> forbidden list". if it is not found in the forbidden list, the method calls the merge. After processing all the equalities, the method moves on the inequalities list. For each inequality, it calls the find

method to obtain the representative values, compares them, and If they are equal, it returns “UNSAT”. If no inequality pair is found to be equal, it returns "SAT" (satisfiable).

```
    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
```

3. SUMMARY OF THE IMPLEMENTATION

For representing all subterm of a formula I use a graph-based data structure in which each node of the graph represents a subterm of the formula. For creating my structure I create two parsers: for the .txt files and for the .smt2 files. I start searching the nested constant and recursively save it and creating the relative node. Step by step I save all the outer functions as new nodes and checking for duplicates. After the creation of the node, I call the function to fill the “ccpar” field. I split the equalities and the inequalities and fill the forbidden list. In the end the solve() function return the satisfiability of the formula.

4. PARSER AND INPUTS

In my implementation there are two Parser class: the txt_parser and the smt_parser.

The txt parser helps me to parse the files with the extension ‘.txt’ and the smt_parser helps me to parse the files with the extension ‘.smt2’.

In my tests I used all CNF formulas:

- First, a ‘.txt’ file with some formulas from the Bradley-Manna textbook. In which each line is a CNF formula to parse. The .txt file must be for example in the following format:

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

  So with only one formula per line, in which each clause is separated by “&” or\and “or” and the function name is indicated by f nad the variables should be separated by “,” ;

  The symbol "=" refers to an equality, the symbol "!=" refers to a disequality.

- Second, some '.smt2' files where each file contains a set of clauses to parse.

In the case in which there are some 'or', I split on the 'or' and if at least one clause is satisfiable then the whole formula is satisfiable.

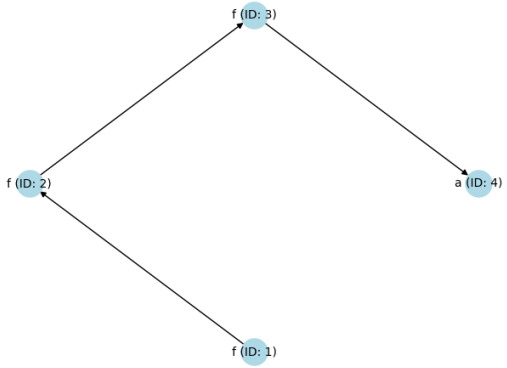
In my implementation there are two different main (main_smt.py and main_smt.py) to facilitate the run of the code in colab and differentiate the outputs.

5. EXAMPLES OF OUTPUT

The algorithm outputs are: the checking formula, the results of the Congruence Closure Algorithm (Sat or Unsat), the execution time, the number of merge and the DAG related to the formula.

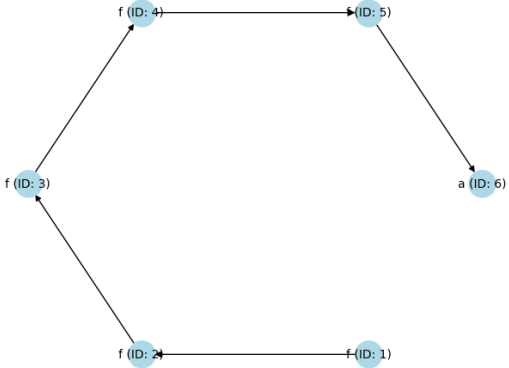
A possible example of a 'sat' formula is: f(f(f(a))) = f(a) & f(f(a)) = a & f(a) != a.

```
*********************************************************************************************
*  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.      *
*                                  for .txt files                                       *
*********************************************************************************************
Checking formula: f(f(f(a))) = f(a) & f(f(a)) = a & f(a) != a
The formula is: SAT
time = 0.00262755199997855
number of merge = 0
---------------------------------------------------------------------------------------------
```



An 'unsat' example is: f(f(f(a))) = a & f(f(f(f(a)))) = a & f(a) != a

```
*********************************************************************************************
*  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.      *
*                                  for .txt files                                       *
*********************************************************************************************
The formula is: UNSAT
time = 0.00293677999979991
number of merge = 2
---------------------------------------------------------------------------------------------
```



An another example, with the forbidden list is: f(a)=f(b) & f(f(f(f(a)))) = a & f(b)!=f(a):

```
*****************************************************************************************
*  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.      *
*                                  for .txt files                                       *
*****************************************************************************************
Checking formula: f(a)=f(b) & f(f(f(f(a)))) = a & f(b)!=f(a)
The formula is: UNSAT -> forbidden list
time = 0.006667282000023533
number of merge = 0
-----------------------------------------------------------------------------------------
```

6. RESULTS OF THE EXPERIMENTS

I do the time and merge experiment only with the .txt file and I found the results below:

Formula:	Sat/Unsat	time	# merge
f(a)=f(b)& f(f(f(f(a)))) = a & f(b)!=f(a)	UNSAT -> fl	0.006266	0
f(a, b) = a & f(f(a, b), b) != a or f(f(f(a))) = a & f(f(f(f(a)))) = a & f(a) != a or f(f(f(a))) = a	SAT	0.011546	3
a=b & b!=a	UNSAT -> fl	0.002679	0
f(a)=f(b) & a!=b	SAT	0.005193	0
a=b & f(a)!=f(b)	UNSAT	0.005628	1
f(f(f(a))) = a & f(f(f(f(a)))) = a & f(a) != a	UNSAT	0.007173	2
f(f(f(a))) = f(a) & f(f(a)) = a & f(a) != a	SAT	0.006582	0
f(a, b) = a & f(f(a, b), b) != a	UNSAT	0.005739	1

7. COMMENTS AND CONCLUSIONS

From what can be seen in the table, the execution time increases as the number of merges increases. In order to prove this, tests with a larger number of clauses would have to be done. Also, as I mentioned above, I've implemented the forbidden list and union with non arbitrary choice as improvements.

One possible solution for the first problem and so a further improvement that could be made, is therefore that to use the QF-UF benchmarks from the the SMT-LIB repository. In order to use them it will have to be added a dnf to cnf converter due to the presence of the numerous 'or' nested.