Planning and Automated Reasoning

Enrico Zorzi

Project of automated reasoning

Implementation of the congruence closure algorithm

June 2023

University of Verona

Introduction

Project Description

In this project, i have implemented a solver based on the congruence closure decision algorithm. The solver is capable of analyzing an SMT-LIB formula or a plain formula, constructing a CC-DAG graph, and solving the satisfiability of the formula. The code is written in Python language utilizing various libraries, including NetworkX, Matplotlib, and PySMT.

Code Structure

The code is divided in 3 sections,parse,CC-DAG and main function.

Parsing the Formula

- `parse_formula(formula, dag)`: This function is responsible for analyzing the formula and extracting the necessary information to construct the CC-DAG graph.

  It uses regular expressions to separate subformulas, handle equalities and inequalities, and identify the nodes present in the formula.

- `translate_string(formula)`: This function translates the formula into a string format that can be understood by the CC-DAG solver, removing unnecessary characters and rewriting certain operators.

- `update_eq_ineq(equalities, inequalities, nodes)`: This function updates the equality and inequality information based on the list of visited nodes.

Constructing the CC-DAG Graph

- `create_graph(dag, nodes)`: This function constructs the CC-DAG graph from the list of visited nodes.It assigns a unique ID to each node and establishes the dependency relationships between nodes. Additionally, it handles function nodes and their arguments.

- `visit(node, nodes)`: This function adds a node to the list of visited nodes.

- `recursive_visit(node, nodes)`: This function performs a recursive visit of the nodes, handling cases of function nodes and breaking down their arguments.

- `sub_visit(node, nodes)`: This function performs a visit of secondary nodes within a function node, handling nested node cases and removing redundant information.

- `visualize_dag(dag)`: This function visualizes the CC-DAG graph using the Matplotlib and NetworkX libraries.

CC_DAG Class

The CC_DAG class represents the CC-DAG solver and contains methods for solving the SMT-LIB formula using the CC-DAG algorithm.

- `CC_DAG`: The constructor of the CC_DAG class initializes the necessary data structures and variables used in the CC-DAG algorithm.
- `solve(formula)`: This method checks the satisfiability of the formula by examining the CC-DAG graph. It iterates over all the inequalities in the formula and checks if the nodes representing their terms belong to different equivalence classes. If an inequality is violated, it returns False.
- `find(node)`: This method finds the representative node of the given node. It recursively traverses the parent pointers until it reaches the representative node and updates the parent pointers along the path to directly point to the representative.
- `union(id1, id2)`: It uses the heuristic to optimize the union operation by always attaching the node with the lower rank to the one with the higher rank.
- `congruent(id1, id2)`: This method checks if two nodes are congruent by comparing their representative nodes.

```
    def congruent(self, id1, id2):
    n1 = self.NODE(id1)
    n2 = self.NODE(id2)
    if (len(n1["args"]) == len(n2["args"])):
        for i in range(len(n1["args"])):
            val1 = self.find(n1["args"][i])
            val2 = self.find(n2["args"][i])
            if val1 != val2:
                return False
        return True
    else:
        return False
```

- `merge(id1, id2)`: This method merges the equivalence classes of two nodes by making one node the parent of the other.

```
    def merge(self, id1, id2):
        a1 = self.find(id1)
        a2 = self.find(id2)
        if a1 != a2:
            pi1 = self.ccpar(id1)
            pi2 = self.ccpar(id2)
            self.union(id1, id2)
            for t1, t2 in list(product(pi1, pi2)):
                if self.find(t1)!=self.find(t2) and self.congruent(t1, t2):
                    self.merge(t1, t2)
            return True
        else:
            return False
```

- `ccpar(id)`: This method returns the parent of the given node.

Main Function

The main function of the code is responsible for reading the input formula from a file, constructing the CC-DAG graph, and solving the satisfiability of the formula. It uses the `parse_formula` function to extract the necessary information from the formula and the `create_graph` function to construct the CC-DAG graph. Finally, it uses the `solve` method of the CC_DAG class to solve the satisfiability of the formula. The main part of the code is shown below:

1. Import necessary libraries and modules.
2. If the script name is a `.smt2` file and exists, perform the following steps:
   (a) Start the timer.
   (b) Use the `SmtLibParser` to parse the SMT-LIB script.
   (c) Translate the formula to a string and remove unnecessary characters.
   (d) Initialize a `CC_DAG`. object
   (e) Parse the formula and create a list of nodes.
   (f) Sort the list of nodes in descending order of length.
   (g) Create a graph using the `CC_DAG` object and the list of nodes.
   (h) Update the equalities and inequalities based on the list of nodes.
   (i) Solve the formula using the `CC_DAG` object and print the result.
   (j) Stop the timer and print the elapsed time.
3. If the script name is a `.txt` file and exists, perform the following steps:
   (a) Start the timer.
   (b) Read the script file line by line.
   (c) If a line contains "or", split the line by "or" and process each sub-formula separately.
   (d) For each sub-formula, create a new `CC_DAG` object, parse the sub-formula, create a list of nodes, sort the nodes, create a graph, update the equalities and inequalities, and solve the sub-formula.
   (e) If a line does not contain "or", create a new `CC_DAG` object, parse the formula, create a list of nodes, sort the nodes, create a graph, update the equalities and inequalities, and solve the formula.
   (f) Print the result for each formula/sub-formula.
   (g) Stop the timer and print the elapsed time.

Use the Solver

To use the solver, provide a .txt or .smt2 file as input to the `main` function. The solver will execute the CC-DAG algorithm on the formula and return the satisfiability result. Additionally, you can visualize the generated CC-DAG graph using the `visualize_dag` function.

N.B. only the .txt can resolve formulas with "or" statements.

Variants implemented

Forbidden List

The forbidden list is implemented by adding the inequalities to a set and checking if the pair of nodes to be merged is in the set. If the pair is in the set, the algorithm returns UNSAT.

Code implementation

```
    def solve(self):
        forbiddenMerges = set()  # set of forbidden list

        for pair in self.inequalities:
            firstId, secondId = pair
            # add to forbidden list
            forbiddenMerges.add((firstId, secondId))

        for pair in self.equalities:
            firstId, secondId = pair
            firstId_find = self.find(firstId)
            secondId_find= self.find(secondId)
            # check if pair is in forbidden list
            if (firstId_find, secondId_find) in forbiddenMerges:
                return "UNSAT"
            self.merge(firstId, secondId)

        for pair in self.inequalities:
            firstId, secondId = pair
            if self.find(firstId) == self.find(secondId):
                return "UNSAT"
        return "SAT"
```

Non arbitrary choice of representative in union

The non arbitrary choice of representative in union is implemented by using the heuristic of attaching the node with the lower rank to the one with the higher rank. This is done by comparing the length of the equivalence classes of the two nodes and attaching the node with the lower length to the one with the higher length.

Code implementation

```
    def union(self, id1, id2):
        n1 = self.NODE(self.find(id1))
        n2 = self.NODE(self.find(id2))
        # Compare the size of the ccpar sets
        if len(n1["ccpar"]) > len(n2["ccpar"]):
            n2["find"] = n1["find"]
            n1["ccpar"].update(n2["ccpar"])
            n2["ccpar"] = set()
        else:
            n1["find"] = n2["find"]
            n2["ccpar"].update(n1["ccpar"])
            n1["ccpar"] = set()
```

Data

I take the .smt2 from the SMT-LIB-benchmarks/QF_UF/-/tree/master/TypeSafe repository.

The .txt file contains some formulas taken from book (The Calculus of Computation by Bradley and Manna) and other formulas created by me.

How to add more data

To add more data, simply add the .smt2 or .txt file to /Test directory.

Format

The .txt file must be in the following format:

```
f(f(f(a))) = f(a) and f(f(a)) = a and f(a) != a
f(x) = f(y) and x != y
f(f(a))=f(b)&f(f(a))!=f(b)
f(a, b) = a and f(f(a, b), b) != a
```

Only one formula per line, the function name indicated by "f" aand the variables should be separated by ",".

The formula with or statement should be in the following format:

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

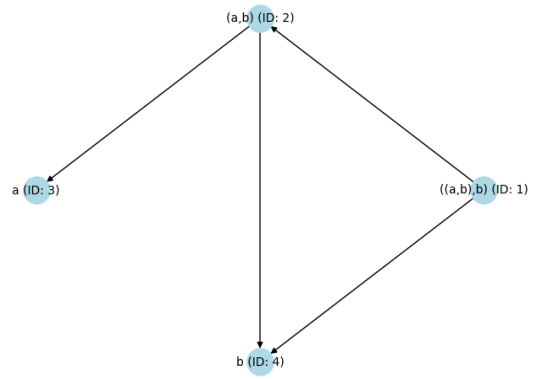
"or" statement is used to separate the sub-formulas.

Time Benchmark

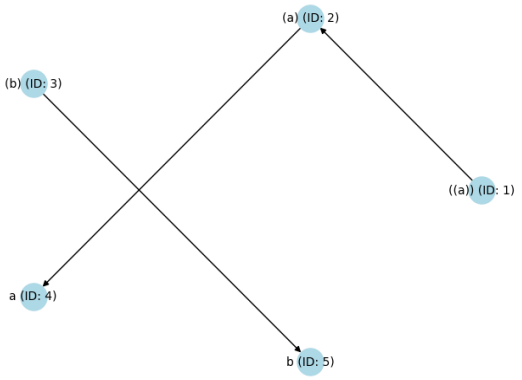
To measure the execution time of the solver, i conduct a benchmark on a set file. The time taken to solve each file is recorded below:

File	Execution Time (seconds)
test1.smt2	0.017
test2.smt2	0.022
test3.smt2	0.013
test4.smt2	0.015
test5.smt2	0.014
test.txt(8 formulas)	0.044

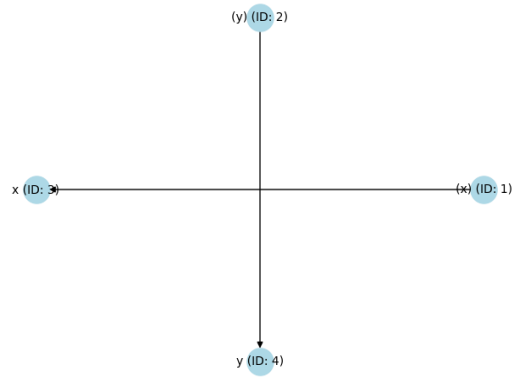
Visualize Graph of .txt file



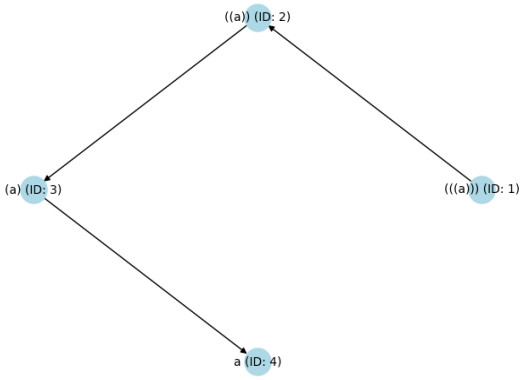
(a, b) = a and ((a, b), b) != a # UNSAT



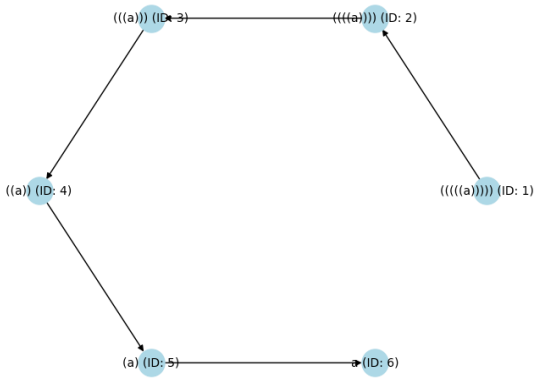
((a))=(b)&((a))!=(b) # UNSAT



(x) = (y) and x != y # SAT



(((a))) = (a) and ((a)) = a and (a) != a # SAT



(((a))) = a and (((((a))))) = a and (a) != a # UNSAT