The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

GENERATION OF INFEASIBLE SUBSETS IN LESS-LAZY SMT-SOLVING FOR THE THEORY OF UNINTERPRETED FUNCTIONS

Lukas Neuberger

Examiners:
Prof. Dr. Erika Ábrahám
PD Dr. Walter Unger

Additional Advisor:
Gereon Kremer, M.Sc.

Aachen, 29.09.2015

Abstract

In the last decade, satisfiability-modulo-theories (SMT) solvers have become very efficient and are used in various real-world applications e.g. in formal verification. SMT solver are combinations of a satisfiability (SAT) solver and a theory solver which determine the satisfiability of formulas from first order logic over some theories. SMT solver for various theories exist, e.g. the theory of linear real arithmetic, nonlinear real arithmetic or linear integer arithmetic. In this thesis, we present a theory solver for the theory of equality logic with uninterpreted functions in lazy (SMT) solving. In our approach, we incrementally calculate the congruence closure over given equalities and check, whether given inequalities are consistent with this closure. While equalities are given to the theory solver, we calculate every congruence and build an equality graph, documenting every equality and congruence in a graph structure. Through the graph structure we are able to generate infeasible subsets in case a conflict between the congruence closure and given inequalities is found. We calculate infeasible subsets by finding a path between nodes in the equality graph. This is done with two heuristics by means of Dijkstra's algorithm for finding shortest paths and Kruskal's algorithm to find a minimum spanning tree. Additionally, these heuristics are used to deduce tautologies, so-called lemmas, aiming at speeding up the solving process. We present three kinds of lemmas. Finally, we test our implementation with a broad range of parameters on a standard benchmark set from SMT-LIB.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Lukas Neuberger
Aachen, den 29. September 2015

Contents

1 Introduction **9**
1.1 Motivation 9
1.2 Related Work 9

2 Preliminaries **11**
2.1 Satisfiability Problem 11
2.2 Satisfiability Modulo Theories Problem 13
2.3 Equality Logic with Uninterpreted Functions 14

3 Theory Solver for Equality Logic with Uninterpreted Functions **17**
3.1 Basic algorithm to check for consistency 18
3.2 Data Structures 19
3.3 Algorithm 25
3.4 Heuristics 30
3.5 Theory Propagation 34

4 Conclusion **37**
4.1 Summary 37
4.2 Experimental Results 37
4.3 Future work 41

Bibliography **43**

Chapter 1

Introduction

The satisfiability (SAT) problem is the problem of deciding whether a given boolean formula is satisfiable, that means, if there exists a satisfying assignment for all its variables. SAT solving is an active research topic in computer science that develops algorithms to solve this problem. In the last decades it has become more and more important, as SAT solvers have become highly efficient. Because boolean logic is not expressive enough for many real-world problems, we combine SAT solvers with theory solvers, decision procedures for conjunctions of theory constraints. This extends the solving capabilities to formulas of first order logic. These combinations of SAT and theory solvers are called satisfiability-modulo-theories (SMT) solver and there exist SMT solver for several first order theories, e.g. linear real arithmetic, nonlinear real arithmetic, linear integer arithmetic or bit vectors. This bachelor thesis discusses a theory solver specifically for equality logic with uninterpreted functions which is integrated into the SMT-RAT library [CKJ+15]. We especially focus on various heuristics to generate infeasible subsets to speed up the solving process.

First we establish a general background knowledge on the SAT and SMT problem, approaches to solve them and the basics of the theory of equality logic with uninterpreted functions in Chapter 2. In Chapter 3, we focus on the needed calculations a theory solver for this theory has to do by means of a simple solving algorithm and explain all important procedures and heuristics of our theory solver. In Chapter 4, we conclude this thesis with a summary, some experimental results and an outlook on future work.

1.1 Motivation

Equality logic with uninterpreted functions is an important logic as there are several real-world use cases. In [BGV01] and [BD94] equality logic with uninterpreted functions is used to abstract data manipulation in a processor when verifying the correctness of its control logic. In [PSS98] it is used to verify that the translation from input source code to the code produced by a compiler is semantically correct.

1.2 Related Work

Historically, equality logic with uninterpreted functions was only considered from a mathematical viewpoint until the mid 1970's. Most notably from that time is Ackermann's work with his reduction method [Ack54]. His method transforms formulas of equality logic with uninterpreted functions to formulas of equality logic by replacing uninterpreted functions with variables and appending constraints to maintain functional consistency. With the emergence of effective theorem provers,

more approaches of solving equality logic with uninterpreted function were developed. Some approaches handle the uninterpreted function directly by calculating the congruence closure over conjunctions of equalities. Shostak proposed a simple method for this approach in [Sho78]. By using syntactic case-splitting, more complex formulas than just conjunctions can be solved by this procedure. But because syntactic case-splitting leads to a much larger formula, this comes with a great cost.

Other solvers that first transform formulas of equality logic with uninterpreted functions into formulas of equality logic with Ackermann's reduction rely more on semantic case-splitting by splitting the finite domain of equality logic formulas rather than the formula itself. Examples for these approaches are [HIKB96] and [HKGB97]. Bryant's reduction [BGV99] is a very similar approach to Ackermann's reduction and was proposed to improve these techniques. [GSZ+03] abstracted from the problem by replacing equalities with boolean variables and building a binary decision diagram (BDD) of this boolean abstraction. The BDD can then be searched for satisfying paths of the boolean abstraction which do not violate the transitivity of equalities. The Sparse method [BV02] is another notable solving approach that also uses a boolean abstraction of equality logic. More details on the history of equality logic with uninterpreted functions in satisfiability checking can be found in [KS08].

The most popular SMT solving approach today combines the boolean abstraction of formulas with directly handling the uninterpreted function by calculating congruence closure. This is done by using a SAT solver to find models for the boolean abstraction and then letting a theory solver check the conjunction of constraints abstracted by the found boolean model for consistency. In this thesis, we present such a theory solver.

Chapter 2

Preliminaries

This chapter introduces the basics of satisfiability solving, satisfiability modulo theories solving and equality logic with uninterpreted functions. The former two are needed to understand the environment in which the theory solver is integrated and the requirements it has to fulfill. Because it is a theory solver for equality logic with uninterpreted functions, the syntax of formulas of equality logic with uninterpreted functions is explained.

Section 2.1 gives an introduction of the satisfiability problem for propositional logic formulas and a rough sketch of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm that is used to solve the problem. Section 2.2 introduces the satisfiability modulo theories problem and different approaches to solve it. The syntax of equality logic with uninterpreted functions is explained in Section 2.3.

2.1 Satisfiability Problem

In the following, we explain the satisfiability problem for boolean formulas.

Definition 2.1.1 (Boolean Formula). *A boolean formula φ can be constructed as follows:*

$$\varphi = (\varphi \wedge \varphi) \mid \neg\varphi \mid x \mid c$$

where $c \in \{True,False\}$ is a constant, $x \in Var(\varphi)$ where $Var(\varphi)$ is the set of all variables in φ.

Syntactic sugar like $(\varphi \vee \varphi)$ or $(\varphi \rightarrow \varphi)$ can be derived from the above construction.

The boolean satisfiability (SAT) problem is the problem of deciding whether a given boolean formula is satisfiable. A boolean formula is satisfiable, if there exists an assignment of its variables so that it satisfies the formula.

A SAT solver utilizes an algorithm to find an assignment

$$\alpha : Var(\varphi) \rightarrow \{True,False\}$$

for all variables in a boolean formula φ that satisfies this formula. Such an assignment is called a model of the formula. The Davis-Putnam-Logemann-Loveland (DPLL) [DLL62] algorithm is a well known SAT solving procedure. In the following we present a simple iterative version of the DPLL algorithm (Algorithm 1).

The algorithm expects a boolean formula in conjunctive normal form as input.

Definition 2.1.2 (Conjunctive Normal Form (CNF)). *A boolean formula varphi is in conjunctive normal form, if it has the following form:*

$$\varphi = \bigwedge_i \bigvee_j l_{ij}$$

where l_{ij} is the j-th literal in the $i-th$ clause. φ is a conjunction of clauses, where each clause is a disjunction of constraints or negated constraints. In the following, constraints and negated constraints are called literals. A boolean formula in conjunctive normal form is satisfiable iff there exists an assignment that satisfies every clause of the formula.

DPLL solves the SAT problem in so called decision levels. In every decision level it assigns an unassigned literal of the boolean formula either to *True* or to *False*. As consequence, it assigns the negation of this literal to the opposite boolean value. If a literal is assigned to *True*, the assignment satisfies all clauses the literal occurs in and these clauses are thereby resolved. A clause is called a conflict clause if all literals are assigned to *False* and the clause is not satisfied by the current assignment. Because an assignment that satisfies the whole formula has to satisfy each clause, this clause causes a conflict indicating that the current assignment cannot be part of a model of the formula. Clauses that do not contain a literal that is assigned to *True* and still contain unassigned literals are called unresolved. If all literals in such an unresolved clause except for one are assigned to *False*, this clause is called a unit clause. The only way we can satisfy a unit clause without reversing decisions is by assigning the unassigned literal to *True*. This is what the DPLL algorithm does after every decision as so called boolean constraint propagation. Because this propagation leads to new assignments, it can enable further propagations.

Example 2.1.1 (Boolean Constraint Propagation). *Let $c : (x_1 \vee x_2)$ be an unresolved clause in a boolean formula φ and x_1,$x_2 \in Var(\varphi)$ be variables in φ. If DPLL decides to assign x_1 to False, c becomes a unit clause because x_2 is the only variable in c that is left unassigned and no other literal in c is assigned True. DPLL propagates and assigns x_2 as True to satisfy c.*

Algorithm 1 An iterative version of the DPLL algorithm for solving the SAT problem for a boolean formula in conjunctive normal form.

```
1: function DPLL(CNFformula)
2:     if !BooleanConstraintPropagation() then return UNSAT
3:     while True do
4:         if !Decide() then return SAT;
5:         while !BooleanConstraintPropagation() do
6:             if !ResolveConflict() then return UNSAT
```

First the algorithm looks for unit clauses in the formula and tries to do boolean constraint propagation (Line 2). This is also referred to as decision level 0, as the algorithm has not yet made any decision. If the boolean constraint propagation in decision level 0 leads to a conflict, this conflict cannot be resolved because there is no decision the algorithm can reverse. The result is, that the formula is not satisfiable and the algorithm returns *UNSAT* (Line 2). Otherwise, the algorithm begins the next decision level and decides the first assignment of a literal (Line 4).

Various heuristics exist for choosing a literal to assign. Choosing the right unassigned literal to assign in a decision level is crucial for the efficiency of this algorithm. In general, it is beneficial, if the unassigned literal that is chosen to be assigned, occurs frequently in unresolved clauses. Assigning a frequently occurring literal might satisfy a lot of clauses or lead to a lot of propagations. So early on, the more far reaching decisions are made.

After every decision, the algorithm invokes boolean constraint propagation again and might find further assignments (Line 5). If a conflict occurs on a higher decision level than decision level 0, the algorithm tries to resolve the conflict (Line 6). It backtracks to the source of the conflict, reversing decisions and propagations. By adding a conflict clause to the formula, which explains the reason for the conflict, it learns, which assignments not to make again. If the source of a conflict is found on decision level 0, the algorithm is not able to resolve the conflict and the formula is unsatisfiable (Line 6). If the algorithm is not able to decide the assignment of another literal, because all literals are already assigned, no conflict has occurred and the formula is satisfiable. The current assignment of the literals is a model of the formula and the algorithm returns SAT (Line 4).

2.2 Satisfiability Modulo Theories Problem

The satisfiability modulo theories (SMT) problem is the problem of deciding whether a given logical formula, expressed in first order logic, is satisfiable with respect to a given theory $\mathcal{T}$. We distinguish between eager SMT solving and lazy SMT solving.

Eager SMT solving Eager SMT solving approaches solve the SMT problem by transforming the original $\mathcal{T}$-formula into an equivalent boolean formula, that is transforming the SMT problem to a SAT problem. After that, a SAT solver is used to decide the problem. Thus, we can re-use the already efficient SAT solver, but we lose the ability to make deductions with knowledge about the underlying theory. This approach is not applicable for all theories. The theory of equality logic with uninterpreted functions and the theory of bit vectors are two example for which there exist eager SMT solving approaches.

Lazy SMT solving Lazy SMT solving approaches abstract the $\mathcal{T}$-formula to its boolean skeleton by substituting every $\mathcal{T}$-constraint by a fresh boolean variable. Then, we use a SAT solver to find a model for this boolean formula. If the SAT solver finds a model, the boolean literals it has assigned are mapped back to their corresponding $\mathcal{T}$-constraints. If a literal is assigned to *True* we map it to its $\mathcal{T}$-constraint, if it assigned to *False* we map it to its negated $\mathcal{T}$-constraint. Then, the SAT solver gives the $\mathcal{T}$-constraints to the theory solver.

The theory solver checks whether the given $\mathcal{T}$-constraints are consistent with respect to $\mathcal{T}$ and returns either satisfiable or unsatisfiable. If it returns satisfiable, a model for the $\mathcal{T}$-formula is found and the SMT problem is solved. Otherwise, if it returns unsatisfiable, the SAT solver remembers that the current assignment of literals is not satisfiable by appending a corresponding clause to the original formula.

The theory solver can also return an explanation on the reasons why the given $\mathcal{T}$-constraints are not consistent in form of a so-called infeasible subset. An infeasible subset contains at least those $\mathcal{T}$-constraints that are not consistent. With this explanation, the SAT solver can append a smaller clause that specifies more precisely the combination of assignments it should not choose again. Generating an infeasible subset is usually associated with a lot of computational work on the side of the theory solver, making theory solver calls more expensive. But infeasible subsets can help reduce the boolean search space for a satisfying assignment

significantly, reducing the remaining assignments the SAT solver has to try out. Additionally to infeasible subsets, the theory solver is allowed to generate lemmas. Lemmas are deductions the theory solver makes with the given constraints and knowledge about the theory. These lemmas have a similar effect as the infeasible subset and are appended as clauses to the original formula.

After receiving an infeasible subset, the corresponding appended clause introduces a boolean conflict in the SAT solver. It has to backtrack and then continues to look for other models of the boolean skeleton extended with clauses for infeasible subsets and lemmas. In case the SAT solver cannot find another model, the SMT problem is solved as unsatisfiable.

Less Lazy SMT Solving The lazy SMT solving approach described above is called full lazy SMT solving, which invokes the theory solver only after finding a complete model for the boolean formula. Less lazy SMT solving approaches additionally invoke the theory solver on partial assignments. After completing all assignments of a decision level, the SAT solver informs the theory solver about the newly assigned boolean literals by adding the corresponding $\mathcal{T}$-constraints to a set of constraints to check for consistency. If a conflict arises and the SAT solver has to backtrack and reverse assignments, it also informs the theory solver about the reversed assignments, removing the corresponding $\mathcal{T}$-constraints from the set. With the lazy approach, a conflict in the underlying theory of the original formula can be found earlier, but the theory solver might be invoked much more often than in the full lazy approach. More details on lazy SMT solving and various approaches can be found in a survey by Roberto Sebastiani [Seb07].

2.3 Equality Logic with Uninterpreted Functions

The theory solver we present in this bachelor thesis is a theory solver for the theory of equality logic with uninterpreted functions (EUF).

Definition 2.3.1 (Formulas of EUF)**.** *Formulas of EUF can be constructed as follows.*

$$\begin{aligned} \textit{Formula } \varphi &:= (\varphi \wedge \varphi) \mid \neg\varphi \mid C \\ \textit{Constraint } C &:= (t = t) \\ \textit{Term } t &:= c \mid x \mid F(t, \ldots, t) \end{aligned}$$

where $c \in D$, is an uninterpreted constant, $x \in Var(\varphi)$ over D is an uninterpreted variable and F is an uninterpreted function $F : D^+ \rightarrow D$. $F(t, \ldots, t)$ is called an uninterpreted function instance of the uninterpreted function symbol F. A constraint is called an equality and a negated constraint is an inequality. In the following, we include negated constraints in the term constraint.

We abstract every constant, variable and function in this logic from its semantics to purely focus on the satisfiability of equalities and inequalities. In this way, we do not have to care about what domain constants and variables are from or what computations are represented by a function and only use their signatures. In principle constants and variables can be from several different domains. However, as soon as the formula is type correct, we can safely assume that all variables and constants are from the same domain, as equalities between variables or constants of different domains are impossible.

The library SMT-RAT that we extend with the theory solver for EUF does some preprocessing to parse formulas of this theory. It flattens uninterpreted function instances by introducing fresh variables for arguments that are uninterpreted function instances.

Example 2.3.1. *Let $C : F(F(x)) = y$ be a constraint of a formula from EUF, where F is an uninterpreted function symbol and x,y are variables. We introduce the new variable s for the uninterpreted function instance argument $F(x)$, substitute the argument and add a new equality to express this substitution. The result is $F(s) = y \wedge F(x) = s$.*

After flattening, we substitute constants by fresh variables and append inequalities to the formula for all pairs of these introduced variables. After parsing, the theory solver has to deal with an adjusted logic.

Definition 2.3.2 (Formulas of EUF in SMT-RAT)**.** *Formulas of the adjusted EUF we are dealing with in SMT-RAT can be constructed as follows.*

$$\begin{aligned} \textit{Formula } \varphi &:= (\varphi \wedge \varphi) \mid \neg\varphi \mid C \\ \textit{Constraint } C &:= (t = t) \\ \textit{Term } t &:= x \mid F(x, \ldots ,x) \end{aligned}$$

where $x \in Var(\varphi)$.

The semantic of EUF is defined by the following axioms of the binary congruence relation $=$. Let t_1,$t_2, \ldots$ be terms and F be an uninterpreted function symbol.

Reflexitvity $t_1 = t_1$

Symmetry $t_1 = t_2 \leftrightarrow t_2 = t_1$

Transitivity $t_1 = t_2 \wedge t_2 = t_3 \rightarrow t_1 = t_3$

Congruence $t_1 = t_2 \wedge t_3 = t_4 \wedge \cdots \rightarrow F(t_1,t_3, \ldots) = F(t_2,t_4, \ldots)$

Reflexivity and symmetry are relevant axioms of this theory, but relatively uninteresting. The main problem when checking a set of equalities and inequalities for consistency is transitivity and functional congruence. In the next chapter we present the needed calculations to calculate transitivities and congruences for a set of EUF constraints.

Chapter 3

Theory Solver for Equality Logic with Uninterpreted Functions

In this chapter, we present an overview of the theory solver for EUF. It builds onto the simple algorithm presented by Shostak [Sho78] which we explain in Section 3.1 and has to fulfill the following three requirements for being used efficiently in lazy SMT solving:

Incrementality: Equalities and inequalities can be added at any time.

Infeasible subsets: If a conflict arises in the theory solver, it has to build an infeasible subsets to explain the conflict.

Backtracking: Equalities and inequalities can be removed at any time. The theory solver has to remember its computation history and backtrack to reverse all changes the addition of an equality or inequality made.

This solver exploits the property of the DPLL algorithm, that it removes assignments of literals in the reverse order in which it assigned them. To use this property, the theory solver stores given equalities and inequalities on separate stacks, so that the respective order is maintained. While adding an equality, its position on the stack is used to indicate the age of the equality. In the following, this numerical position of an equality in the stack is called its *Stage*, starting with 1. This *Stage* of every added equality is also used to mark every effect the addition of the equality has on various data structures to enable easy backtracking. As equalities are always removed from the top of the stack, the equality with the currently highest *Stage* is removed and we can reverse every effect on data structures marked with the corresponding *Stage*. Additionally to the given equalities we calculate equalities that hold because of functional congruence. These equalities are also assigned a *Stage*, the same as the equality that causes the calculation of this congruence. Furthermore, these equalities are assigned a *Sub Stage*, to indicate, that they are slightly older than the equality that caused them.

This chapter is structured as follows. First, we explain the needed calculations for a consistency check of constraints from EUF on the basis of a simple algorithm performing that task in Section 3.1. Then, we introduce the important data structures used in the theory solver in Section 3.2. We use a *union-find* data structure to track the equivalence classes of terms, a custom data structure called *computation graph* to document changes in the equivalence classes and a graph structure called *equality graph* to generate infeasible subsets. Section 3.3 describes the actual procedures of the theory solver for adding and removing constraints

from EUF and for checking the given constraints for consistency. In Section 3.4, we present two heuristics to generate infeasible subsets, and in Section 3.5, we present three kinds of lemmas this theory solver can generate.

3.1 Basic algorithm to check for consistency

In this section, we present the algorithm of Shostak [Sho78] that checks a conjunction of EUF-constraints for consistency. Given a set of constraints from EUF, the equalities of this set define a relation $=$, as they have to be contained in this relation. To check a set of constraints for consistency, we have to calculate the congruence closure over the relation $=$. The congruence closure over the relation $=$ is the smallest congruence relation that contains $=$.

Algorithm 2 Simple algorithm to check a set of constraints of EUF for consistency

1: **function** CHECK(*ConstraintSet*)
2: $EC \leftarrow \{\{t\} \mid t$ occurs as term in an (in)equality in $ConstraintSet\}$
3: **for all** $t = t'$ in *ConstraintSet* with $[t] \neq [t']$ **do**
4: $EC \leftarrow (EC \setminus \{[t],[t']\}) \cup \{[t] \cup [t']\}$
5: **while** exists $F(\bar{t}),F(\bar{t}')$ in *ConstraintSet* with $[\bar{t}_i] = [\bar{t}'_i]$ $(i \in \{1,\dots,n\})$ and $[F(\bar{t})] \neq [F(\bar{t}')]$ **do**
6: $EC \leftarrow (EC \setminus \{[F(\bar{t})],[F(\bar{t}')]\}) \cup \{[F(\bar{t})] \cup [F(\bar{t}')]\}$
7: **for all** $t \neq t'$ in *ConstraintSet* **do**
8: **if** $[t] = [t']$ **then return** UNSAT
9: **return** SAT

To calculate the congruence closure, we first calculate the equivalence closure over the relation $=$. This is the smallest equivalence relation that contains $=$. To do this, we first define a new equivalence class for every term occurring in the constraints and EC as the set of these equivalence classes (Line 2). Initially, every term is the only element in its equivalence class. For better readability, $[t]$ represents the equivalence class t is in and $\bar{t}$ a group of terms $t_1, \dots, t_n$. Now, we iterate over every equality in the set of constraints and merge the equivalence classes of the left hand and right hand term of every equality (Lines 3-4). By doing this, terms that are equal by transitivity also end up in the same equivalence class. The resulting equivalence classes represent the equivalence closure over $=$ for the given equalities, as every pair of elements contained in the same equivalence class has to be contained in the equivalence closure by transitivity.

Example 3.1.1. *Let a,b,c be variables and $c_1 : a = b, c_2 : b = c$ be equalities from EUF. At the beginning, every variable is in its own equivalence class $\{a\}$,$\{b\}$,$\{c\}$. First, we consider c_1 and merge the equivalence classes of a and b: $[a] \cup [b]$. This results in the equivalence classes $\{a,b\}$,$\{c\}$. Next, we consider c_2 and merge the equivalence class of b and c: $[b] \cup [c]$. This results in the equivalence class $\{a,b,c\}$. As we can see, a and c are in the same equivalence class by transitivity.*

To calculate the congruence closure over $=$, we need to check all pairs of terms that are uninterpreted function instances of the same uninterpreted function symbol and not in the same equivalence class. We have to check for every one of these pairs, whether all their arguments at the same position are in the same equivalence class (Line 5). If this is the case, this pair of uninterpreted function instances is equal by functional congruence and we merge their equivalence classes (Line 6). The problem is that, by merging equivalence classes, other pairs of uninterpreted function instances now might have become congruent too. So, we have to start again to compare all pairs of uninterpreted function instances of the same uninterpreted function symbol. We continue doing this, until every pair of uninterpreted

function instances with equivalent arguments at all positions are in the same equivalence class. Now, the equivalence classes represent the congruence closure over the relation = defined by the given equalities.

Next, we check for all inequalities if they are consistent with the equivalence classes (Line 7). If the left hand and right hand term of an inequality are in the same equivalence class we have found a conflict and the given set of constraints is not consistent (Line 8). If no conflict is found, the set of constraints is consistent (Line 9).

3.2 Data Structures

In the following, we will define the essential data structures we use to fulfill the three requirements mentioned above with our algorithm.

3.2.1 Union-Find Structure

A *union-find* structure is a data structure that represents disjoint sets. Every set is identified by a representative element of the set. The data structure has three operations:

MakeSet(x) takes one element x that is not yet contained in any other set and creates a new set with x as its only element. Therefore, x is also the representative of this new set.

Union(x,y) joins the sets in which x and y are contained.

Find(x) returns the representative of the set in which x is contained. By comparing the results of `Find`(x) and `Find`(y), we can determine whether x and y are in the same set.

We use the union-find structure to represent the equivalence classes of terms of added equalities. Each set in the *union-find* structure represents an equivalence class. With this structure we can join whole equivalence classes and check whether two elements are in the same equivalence class.

An efficient way to implement this kind of data structure is using a directed forest $G = (V,E)$, a disjoint set of trees. Each set is represented by a tree, where each node $v \in V$ is labeled with an element and is connected to a parent node $v_p \in V$ by an edge $(v,v_p) \in E$. The root $v_r \in V$ of such a tree has an edge (v_r,v_r) looping to itself and its label is the representative for the set. `MakeSet`(x) creates a new tree by adding a new node $V := V \cup \{v_x\}$ labeled with the element x and adding a looping edge $E := E \cup \{(v_x,v_x)\}$ to mark it as the root of the new tree. When calling `Find`(x), we trace the way from the node $v_x \in V$ labeled with x to the root node $v_r \in V$ of the tree it is in and return its label as representative of the set. Joining two sets with `Union`(x,y) can be done by attaching the root node $r_x \in V$ of the tree the first parameter is in to the root node $r_y \in V$ of the tree the second parameter is in. We do this by removing the looping edge from r_x to itself and adding an edge from r_x to r_y: $E := (E \setminus \{(r_x,r_x)\}) \cup \{(r_x,r_y)\}$.

This tree structure additionally needs a mapping from an element to its parent. In case of continuous integer elements, the forest can be implemented as an array where the location of the element stores its parent element.

Example 3.2.1 (Union-Find as Forest). *Let a,b,c be elements and $v_a, v_b, v_c \in V$ their respective nodes in the graph $G = (V,E)$. After calling `MakeSet` on every element to initialize the data structure, the trees of G look like Figure 3.1. First, we join the sets containing a and b by calling `Union`(a,b) (Figure 3.2). Because*

v_a and v_b are the roots of their respective tree, we just remove the looping edge from v_a and add an edge from v_a to v_b. Then, we join the sets containing a and c by calling `Union(a,c)` (Figure 3.3). The root of the tree, in which v_a is in, is v_b, so, v_b's looping edge is removed and an edge from v_b to v_c is added. Now, if we call `Find` on any of the elements, we traverse the tree up to the root, that has an edge to itself, and return its label as representative. In the example, every `Find` call would traverse to the root node v_c and return c as representative.

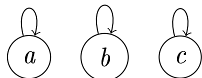

Figure 3.1:
After initialization

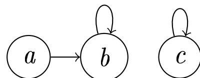

Figure 3.2:
After `Union`(a,b)

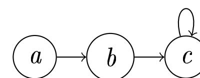

Figure 3.3:
After `Union`(a,c)

Optimizing

The naive implementation described above has a worst-case run time of $\mathcal{O}(n)$ per `Union` and `Find` operation because the trees are not balanced and can deteriorate into singly linked lists. There are two optimizations that vastly improve the run time.

The first optimization is called *union by rank* because it considers the depth of trees. Every node is additionally labeled with a rank that states the depth of the sub tree of that node. When a node is created with `MakeSet`, it starts with a rank of 0. In the `Union` operation, the smaller tree, determined by the rank of the root, is always attached to the bigger tree and, thus, does not increase the depth of the tree. Only if both roots of the trees have the same rank, attaching one root to the other increases the rank of that root by one and the tree's depth also increases by one. Thus, trees cannot deteriorate into lists and the worst-case run time of the `Union` and `Find` operations becomes $\mathcal{O}(\log n)$.

Example 3.2.2 (Union by Rank)**.** *We us the same procedure as in Example 3.2.1, this time applying the optimization* union by rank*. The elements a,b,c are initialized with `MakeSet` (Figure 3.4). The number below the node is its corresponding rank. First, we join a and b (Figure 3.5). As both roots v_a and v_b of the respective trees have the same rank, we, again, just attach the root v_a of the first parameter to the root v_b of the second parameter, increasing its rank by 1. After that, we join a and c (Figure 3.6). Now, the root v_b of the tree containing v_a has a higher rank than v_c, so we just attach v_c to v_b, without increasing the rank of v_b. The tree is now balanced and `Find` and `Union` calls have a shorter amortized run time.*

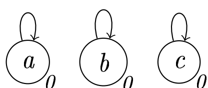

Figure 3.4:
After initialization

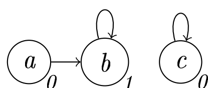

Figure 3.5:
After `Union`(a,b)

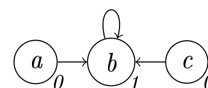

Figure 3.6:
After `Union`(a,c)

The second optimization is called *path compression*. Every time we trace the path from a node to the root of the tree, we can attach every node that is traversed, and thereby its sub tree, directly to the root. Thus, the tree is flattened and future `Union` and `Find` operations are speeded up.

Example 3.2.3 (Path Compression)**.** *This example uses a similar but bigger* union-find *structure then the resulting structure of Example 3.2.1 and only uses* path compression *as optimization. We will examine what a `Find` call does when called on an element deep in a tree. We begin with the tree in Figure 3.7. When we call `Find(a)`, we traverse the path $a \to b \to c \to d \to e$ and attach every node*

we traverse to the root e. After the `Find` *call, the tree is compressed (Figure 3.8)*

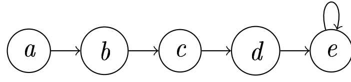

Figure 3.7: Before `Find`(a)

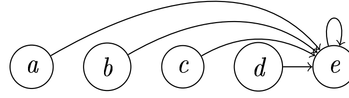

Figure 3.8: After `Find`(a)

When using both optimizations together, the data structure still yields a worst-case run time of $\mathcal{O}(\log n)$ for `Union` and `Find`, but they achieve an amortized run time per `Union` and `Find` of $\mathcal{O}(\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackermann function $n = A(x,x)$. This was shown by R. Tarjan in 1975 [Tar75].

Due to the implementation of the *union-find* structure, optimized for a fast run time, reversing a `Union` operation of the *union-find* structure cannot be done with the structure itself. To be able to do this, we embed the *union-find* structure into the custom data structure *computation graph*. This data structure has full access to the *union-find* structure and is allowed to manipulate it.

3.2.2 Computation Graph

The *computation graph* is an undirected forest $G = (V,E)$, a disjoint set of trees. Every node of the graph is labeled with

$$L : V \rightarrow (T,S)$$

where T is a set of terms from EUF and S is a numerical *Stage*. A node $v \in V$ has a maximum of one edge $(v,v_{par}) \in E$ to another node v_{par} labeled with a higher *Stage* and can have edges to multiple nodes $v_{pre} \in V_{pre} \subset V$ with a lower *Stage*. v_{par} is called the parent of v and V_{pre} is the set of predecessors of v. The root of a tree is the node labeled with the highest *Stage*.

The *computation graph* represents the computation history of the *union-find* structure. Therefore, it represents the computation history of the equivalence classes. A node represents an equivalence class at a specific *Stage*. So, the roots of all trees represent the current equivalence classes, exactly the same as the *union-find* structure. The predecessors of a node $v \in V$ represent the previously disjoint equivalence classes that were unified to form the equivalent class represented by the node v.

Initially, the graph has a node v_t for every occurring term t in the given equalities. Each initial node v_t is labeled with $L(v_t) = (\{t\},0)$. The *computation graph* has three main operations:

`Union`(t_1,t_2,s) calls the `Union`(t_1,t_2) operation on the *union-find* structure and finds the roots r_{t_1} and r_{t_2} of the trees which contain the nodes v_{t_1} and v_{t_2} in the *computation graph*. The roots are labeled with $L(r_{t_1}) = (T_1,S_1)$ and $L(r_{t_2}) = (T_2,S_2)$.

- In case neither $s = S_1$ nor $s = S_2$ holds, we add a new node r_{new},

  $$V := V \cup \{r_{new}\},$$

  and add edges from r_{t_1} and r_{t_2} to this new node,

  $$E := E \cup \{(r_{t_1}, r_{new}),(r_{t_2},r_{new})\}.$$

  We label r_{new} with $L(r_{new}) = (T_1 \cup T_2, s)$.

- In case only $s = S_1$ holds and $s = S_2$ not, we add an edge from r_{t_2} to r_{t_1},

$$E := E \cup \{(r_{t_2}, r_{t_1})\},$$

and append T_2 to T_1,

$$T_1 := T_1 \cup T_2.$$

- In case both $s = S_1$ and $s = S_2$ hold, we merge the two nodes r_{t_1} and r_{t_2}. We do this by removing the node r_{t_2},

$$V := V \setminus \{r_{t_2}\},$$

unifying their labeled sets

$$T_1 := T_1 \cup T_2$$

and replacing all edges to r_{t_2} with edges to r_{t_1}.

AreEquivalent(t_1,t_2) compares the result of the Find(t_1) and Find(t_2) operations on the *union-find* structure. If both operations return the same representative it returns *True*, otherwise *False*.

Reverse(s) removes all nodes $v \in V$ labeled with $L(v) = (T,S)$ where $S = s$. For every removed node v we collect the predecessor set V_{pre} and partition the the corresponding equivalence class of v in the *union-find* structure into the equivalence classes represented by the predecessors V_{pre}. For every predecessor node $v_{pre} \in V_{pre}$ we call MakeSet on every element in the labeled set and, afterwards, unify all elements of this set by calling the Union operation on the *union-find* structure accordingly.

By documenting the computation history of the *union-find* structure, the *computation graph* enables us to reverse Union operations, with one limitation. We cannot reverse individual Union operations, but rather all Union operations on the currently highest Stage. This is exactly what we want when removing an equality from the top of the stack. We want to reverse all effects the addition of this equality had on the *union-find* structure.

Example 3.2.4 (Computation Graph). *We use the same procedure as in Example 3.2.2. Let a,b,c be elements. The* computation graph *after initialization for a,b,c is shown in Figure 3.9 with the internally initialized* union-find *structure underneath. Similar to Example 3.2.2, we first join a and b on Stage 1, by calling Union(a,b,1). Next, we join b and c with Stage 2, so, we call Union(b,c,2). As we can see in Figure 3.10 and Figure 3.11, always choosing a higher Stage allows us to build up the* computation graph *in layers. Now we reverse the last Stage by calling Reverse(2). The first step shown in Figure 3.12 removes all nodes from Stage 2 in the* computation graph*. The second step shown in Figure 3.13 re-initializes the elements from the predecessors of the removed node by calling MakeSet on the* union-find *structure. The second step shown in Figure 3.14 reunifies the elements of the predecessors. After these three steps, the* computation graph *and the* union-find *structure are reversed to their state before Union(b,c,2) was called.*

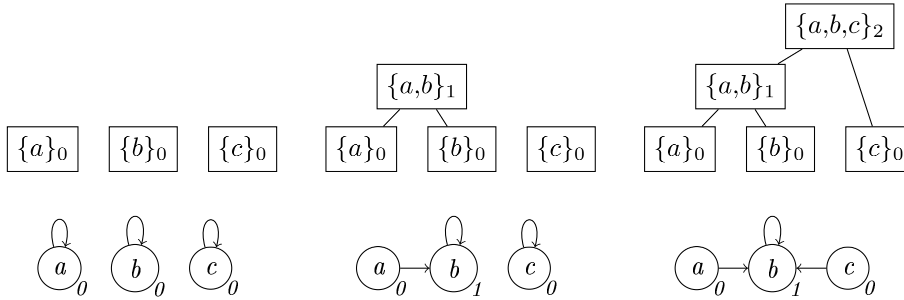

Figure 3.9:
After initialization

Figure 3.10:
After `Union`(a,b,1)

Figure 3.11:
After `Union`(b,c,2)

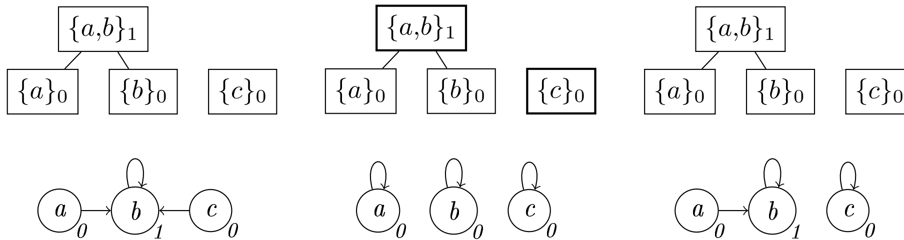

Figure 3.12:
Remove nodes on *Stage* 2

Figure 3.13:
Reinitialize elements in predecessors (bold)

Figure 3.14:
Reunify elements in predecessors

Furthermore, the represented computation history helps us in processing the congruence closure. It allows us to look up from which formerly disjoint equivalence classes an equivalence class is composed of. We will elaborate on this later.

3.2.3 Equality Graph

The *equality graph* documents all given equalities and congruences between uninterpreted function instances.

Definition 3.2.1 (Equality Graph). *Given a set of equalities Φ from EUF, the corresponding* equality graph *is an undirected graph $G = (V, E_{explicit} \cup E_{implicit})$. Each node $v \in V$ is labeled with a term occurring in Φ, each edge $e_{explicit} \in E_{explicit}$ corresponds to an equality in Φ and is called an explicit edge and each edge $e_{implicit} \in E_{implicit}$ corresponds to an equality implied by functional congruence and is called an implicit edge.*

Example 3.2.5 (Equality Graph). *Let $\Phi = \{(x = y),(F(x) = z),(F(y) = z)\}$ be a set of equalities. The* equality graph *for Φ is shown in Figure 3.15.*

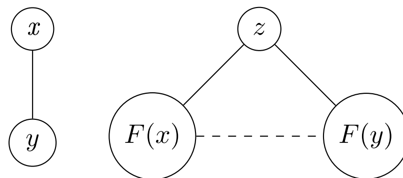

Figure 3.15: *Equality graph* for $\Phi = \{(x = y),(F(x) = z),(F(y) = z)\}$. Explicit edges are shown as solid edge, implicit edges as dashed edges.

The *equality graph* has three operations. Let t_1,t_2 be terms from EUF.

`AddEdge`(t_1,t_2, *IsImplicit*, s) if there is no node $v \in V$ that is labeled with a term t_1 or t_2, we add the according node to V. Then, we add an edge between the nodes labeled t_1 and t_2. If the boolean parameter *IsImplicit* is *True*, the edge is added to $E_{implicit}$, otherwise to $E_{explicit}$. In both cases, we

label the edge with the *Stage* s. If the edge is implicit, we extend the *Stage* label with a *Sub Stage*. This *Sub Stage* is increased for every implicit edge added on the same *Stage*.

`RemoveEdges`(s) removes all edges labeled with a *Stage* equal to s.

`Explain`(t_1,t_2) uses the *equality graph* to explain why t_1 and t_2 are equivalent by returning an explanation in form of a set of equalities occurring in Φ that implies this equivalence.

A path in an *equality graph* $G = (V, E_{explicit} \cup E_{implicit})$ from a node $v_1 \in V$ to a node $v_2 \in V$ represents such an explanation for the equivalence of the corresponding terms of v_1 and v_2. By collecting all equalities represented by the edges used in this path, we get a set of equalities explaining the equality. This set might contain equalities that only hold because of congruence and do not occur in Φ. Because we want to use the `Explain` operation to generate infeasible subsets and infeasible subsets are subsets of the given equalities and inequalities in Φ, we need to remove all equalities in the explanation that are not contained in Φ. We do this by replacing them with an explanation of why their congruence holds. Congruence can only occur between terms of uninterpreted function instances of the same function symbol when all their pairs of $i-$th arguments are equivalent, where $i \in 1, \ldots, n$ and n is the number of arguments needed by the uninterpreted function. Such a congruence can be explained by explaining the equality of each of these pairs of arguments. When explaining a congruence, we need to make sure not to use implicit edges in the graph that represent congruences which depend on the congruence we currently want to explain. To avoid this circular dependency, we restrict the graph for the following `Explain` operations until the congruence is fully explained. For this, we need the label *Stage* s and *Sub Stage* t of the corresponding implicit edge of the congruence we want to explain. We restrict the equality graph $G = (V, E_{explicit} \cup E_{implicit})$ to a graph $G' = (V, E_{explicit} \cup E'_{implicit})$ where

$$E'_{implicit} := \{e \in E_{implicit} \mid \ Stage(e) < s \vee (Stage(e) = s \wedge Sub\ Stage(e) < t)\}.$$

Example 3.2.5 (continued). *[Explaining an Equality]*

When we call `Explain`$(F(x),F(y))$ *on the* equality graph *shown in Figure 3.15, we could use the implicit edge between the two nodes labeled with $F(x)$ and $F(y)$ as path. The explanation E generated from this path would be the set $E := \{F(x) = F(y)\}$. The equality $F(x) = F(y)$ only holds because of congruence and is not contained in Φ. We now need to find an explanation for this congruence by explaining the equality of x and y. The explanation generated for the equality of x and y is the set $E' := \{x = y\}$. We replace the congruence in the original explanation E and append the explanation for the congruence E':*

$$E := (E \setminus \{F(x) = F(y)\}) \cup E'.$$

The resulting explanation is $E := \{x = y\}$. As all equalities in this explanation are contained in Φ, this explanation can be used to generate an infeasible subset.

Another possible explanation for the equality of $F(x)$ and $F(y)$ can be found in the equality graph *with the path from $F(x)$ to z, and from there to $F(y)$. This path only contains explicit edges and, therefore, all representing equalities are contained in Φ. $E := \{F(x) = z, z = F(y)\}$ is another valid explanation for the equality of $F(x)$ and $F(y)$ to be used in an infeasible subset.*

Explanations for a congruence can themselves contain equalities that only hold because of congruence. We have to recursively explain these congruences. Because any congruence must have originated only from the equalities in Φ, we are always able to generate an explanation only containing equalities from Φ.

With the *equality graph*, all equalities between terms in the same equivalence class can be explained. Often, more than one path between two nodes exists, so there are different ways to explain an equality and we are flexible as to how we choose an explanation e.g. by labeling the edges with weights and using various graph algorithms to find paths. We use this procedure to generate infeasible subsets and lemmas with various heuristics. In Section 3.4 we elaborate on the heuristics we use in the `Explain` operation.

3.3 Algorithm

The theory solver used in lazy SMT solving has to be able to receive new equalities and inequalities, remove equalities and inequalities and to check the current equalities and inequalities for consistency at any time. This means that we need to define the methods `Add`, `Remove` and `Check`. Theses methods are invoked by the SAT solver when needed. Initially, the theory solver receives some constraints through the `Add` method one by one and then has to check their consistency via the `Check` method. If the `Check` method determines the constraints to be consistent, the SAT solver can continue with solving the boolean skeleton. That means adding further constraints or removing constraints with the `Remove` method, in case it encounters a boolean conflict. If the `Check` method determines the constraints to be inconsistent, it calculates and returns an infeasible subset. The SAT solver adds this infeasible subset as clause to the original formula which introduces a conflict in the boolean skeleton. This forces the SAT solver to backtrack and remove constraints via the `Remove` method. After that, it continues to solve the boolean skeleton.

In the following, we present a simplified version of the algorithm for the `Add`,`Remove` and `Check` methods. The global variables *AddedEqualities* and *AddedInequalities* represent the separate stacks for added equalities and inequalities, the global variable *UFCounter* represents a counter for every uninterpreted function instance that counts the number of occurrences of an instance in the added equalities.

3.3.1 Adding Constraints

In this section, we present the procedure `Add` that is invoked, when the SAT solver wants to give a newly assigned constraint to the theory solver. The Algorithm 3 receives one constraint at a time.

First, we distinguish between equalities and inequalities (Line 2). Inequalities are collected on the stack *AddedInequalities* for remembering the order in which they were added (Line 17). Adding an equality is more difficult. Similar to inequalities, we first push an equality on the separate stack *AddedEqualities* (Line 3). Then, we add the representing explicit edge for the equality in the *equality graph* (Line 7), labeled with the current *Stage* obtained from the size of the stack *AddedEqualities* (Line 4).

The new equality might force us to update our current representation of the equivalence classes or further update the *equality graph* of the given equalities. This happens either if we add an equality that unifies two equivalence classes or if we add an equality with at least one uninterpreted function instance that has not yet occurred in the given equalities and is congruent to at least one other uninterpreted function instance that already occurred in another added equality. In the first case (Line 8), we unify the equivalence classes of the terms and document the computation history by calling the `Union` operation on the *computation graph* with the terms and the *Stage* as arguments (Line 9). In the second case, we first have to detect whether an uninterpreted function instance in the added equality has already occurred in another added equality. We use a separate counter for

Algorithm 3 Adds a constraint to the set of constraints the theory solver has to check

```
1: function Add(Constraint)
2:     if IsEquality(Constraint) then
3:         AddedEqualities.push(Constraint)
4:         Stage ← AddedEqualities.Size()
5:         Lht ← Constraint.LeftHandTerm
6:         Rht ← Constraint.RightHandTerm
7:         EqualityGraph.AddEdge(Lht, Rht, False, Stage)
8:         if Not ComputationGraph.AreEquivalent(Lht,Rht) then
9:             ComputationGraph.Union(Lht, Rht, Stage)
10:        for all t in {Lht,Rht} do
11:            if IsUF(t) then
12:                UfCounter.Increase(t)
13:                if UfCounter.Get(t) = 1 then
14:                    CalculateCongruenceForUF(t,Stage)
15:        CalculateAllCongruence(Stage)
16:    else
17:        AddedInequalities.Push(Constraint)
```

every uninterpreted function instance to count how often an instance occurs in the added equalities (Line 12). This is done within the variable *UFCounter*. If an instance is added for the first time, having a count of one (Line 13), we have to check all other occurring uninterpreted function instances of the same function symbol in order to see whether a congruence between them and the new instance holds from previously added equalities (Line 14, Algorithm 4).

If any of the previous operations joins at least two equivalence classes, we further have to check for newly formed congruences (Line 15, Algorithm 5). Not only do we calculate the congruence closure, but we also find every congruence that has formed to complete the equality graph. New congruence can only originate from the union of equivalence classes, because this is the only way for arguments of a pair of uninterpreted function instances to become equal.

Finding congruences for newly added uninterpreted function instances

We start by iterating over all uninterpreted function instances that occur in *Added-Equalities* of the same function symbol as *UF* (Lines 2-3). For each of these uninterpreted function instances, we check, whether congruence between them and *UF* holds (Lines 5-7). If such a congruence is found, we can use the found instance *CongruentUF* to find all other congruences with *UF*. Because *CongruentUF* was added in an earlier `Add` call, its congruence is already calculated and represented by implicit edges in the *equality graph*. We just have to add implicit edges between *UF* and all other nodes connected to *CongruentUF* as well as between *UF* and *CongruentUF* (Lines 11-15). Every added implicit edge is labeled with the current *Stage*. Because *UF* is newly added and has not yet occurred in another added equality, it cannot yet be in the same equivalence class as *CongruentUF* and the other instances it is congruent to. So we update the *union-find* structure and the *computation graph* accordingly (Line 16).

Calculating congruence closure and finding all congruences

To find all newly formed congruences, we first have to find every pair of equivalence classes that were joined this *Stage* and insert them into a set *PairsToCheck*. Through the *computation graph* we can figure out, which equivalence classes were

Algorithm 4 Finds all congruences for a newly added uninterpreted function instance

```
1: function CalculateCongruenceForUF(UF,Stage)
2:     for all subexpression t in equalities of AddedEqualities do
3:         if t is function and t.Symbol = UF.Symbol then
4:             Equal ← True
5:             for all arguments (a,b) in t,UF at the same position do
6:                 if !ComputationGraph.AreEqual(a,b) then
7:                     Equal ← False
8:             if Equal then
9:                 CongruentUF ← t
10:                Break
11:    if CongruentUF then
12:        Edges ← EqualityGraph.OutgoingEdges(CongruentUF)
13:        for all  implicit edges (t,v) in Edges do
14:            EqualityGraph.AddEdge(UF, v, True, Stage)
15:        EqualityGraph.AddEdge(UF, CongruentUF, True, Stage)
16:        ComputationGraph.Union(UF, CongruentUF, Stage)
```

joined this *Stage* as the corresponding nodes are labeled with this *Stage* (Line 3). For every one of these equivalence classes, we can also find the previously disjoint equivalence classes that were joined to form this equivalence class by finding all predecessors of the corresponding node in the *computation graph* (Line 5). Because the *computation graph* does not document every exact `Union` operation we have to insert every possible pair of equivalence classes represented by the predecessors of a joined equivalence class into the set *PairsToCheck* (Line 7).

We compute every pair (A,B) of equivalence classes in *PairsToCheck*(Line 8). For each equivalence class in this pair we collect all uninterpreted function instances that have at least one argument from that equivalence class in separate sets UF_A, UF_B (Lines 9-10). We now check for each pair of uninterpreted function instances $(a,b) \in UF_A \times UF_B$ if congruence between a and b holds (Lines 11-12). If a new congruence is found, we add an implicit edge to the *equality graph* labeled with the current *Stage* (Line 13). In case the congruence is between terms that are not yet in the same equivalence class, we update the *union-find* structure and the *computation graph* accordingly (Lines 14-15). Through new found congruence further equivalence classes might be joined, which might again lead to more congruences. So, for these newly joined equivalence classes, we again insert all possible pairs with other predecessors into the set *PairsToCheck*(Lines 16-19).

If the set *PairsToCheck* is completely processed, we know that we have found all congruence introduced through the new equality, the *union-find* structure correctly represents the congruence closure of all added equalities and the *equality graph* contains all possible implicit edges.

3.3.2 Checking for Consistency

In this section, we present the procedure `Check` that is invoked, when the SAT solver wants the theory solver to check the currently assigned constraints for consistency. The Algorithm 6 returns *True*, if the constraints are consistent, or *False* otherwise. As we update our representation of the congruence closure with each `Add` operation, we can assume that the *union-find* structure correctly represents the congruence closure of the given equalities. Now, we can check all added inequalities for consistency with the equalities by iterating over the inequalities (Line 3), checking via the *union-find* structure whether their left hand and right hand terms are in the same equivalence class (Line 6).

Algorithm 5 Calculates congruence closure for the current *Stage* and finds all newly formed congruences

```
 1: function CalculateAllCongruence(Stage)
 2:     PairsToCheck ← {}
 3:     EqClasses ← ComputationGraph.FindEqClasses(Stage)
 4:     for all EqClass in EqClasses do
 5:         FormerEqClasses ← ComputationGraph.GetPre(EqClass)
 6:         for all (A,B) where A,B in FormerEqClasses do
 7:             PairsToCheck ← PairsToCheck∪{(A,B)}
 8:     for all (A,B) in PairsToCheck do
 9:         UF_A ← AllUFContainingArgumentFrom(A)
10:         UF_B ← AllUFContainingArgumentFrom(B)
11:         for all (a,b) where a in UF_A, b in UF_B do
12:             if areCongruent(a,b) then
13:                 EqualityGraph.AddEdge(a, b, True, Stage)
14:                 if Not ComputationGraph.AreEquivalent(a,b) then
15:                     ComputationGraph.Union(a, b, Stage)
16:                     EqClass ← ComputationGraph.GetNewEqClass()
17:                     FormerEqClasses ← ComputationGraph.GetPre(EqClass)
18:                     for all (A,B) where A,B ∈ FormerEqClasses
                                and (A,B) ∉ PairsToCheck do
19:                         PairsToCheck ← PairsToCheck∪{(A,B)}
```

Algorithm 6 Checks the current set of constraints for consistency

```
 1: function Check
 2:     IsConsistent ← True
 3:     for Inequality in AddedInequalities do
 4:         Lht ← Inequality.LeftHandTerm
 5:         Rht ← Inequality.RightHandTerm
 6:         if ComputationGraph.AreEquivalent(Lht,Rht) then
 7:             IsConsistent ← False
 8:             Explanation ← EqualityGraph.Explain(Lht, Rht)
 9:             InfeasibleSubset ← Explanation ∪ {Inequality}
10:             InfeasibleSubsets ← InfeasibleSubsets ∪ InfeasibleSubset
11:     return IsConsistent
```

If we find an inequality, whose terms are in the same equivalence class, we have found a conflict and generate an infeasible subset to return to the SAT solver. We do this by finding an explanation for the equality of the left hand and right hand terms of the inequality in the *equality graph* (Line 8). The inequality is appended to the found explanation (Line 9) and we have generated an infeasible subset for the conflict. Because there might be multiple conflicts, we add the infeasible subset to a global set of infeasible subsets *InfeasibleSubsets* (Line 10) and continue to iterate over the remaining inequalities in order to look for more conflicts. If we find at least one conflict, the theory solver returns *False* to indicate that the given constraints are not consistent. If we find no conflict, it returns *True* to indicate that the given constraints are consistent.

Again the problem arises that an uninterpreted function instance in an inequality might not occur in the added equalities and its congruence to other uninterpreted function instances is not calculated. In this case, we need to look for congruence between the non occurring instance and all occurring instances in the equalities. For every not occurring instance we again only look for one congruence. If a congruence is found, the occurring instance serves as representative for the non

occurring instance to check for equivalence between the terms of the inequality. If no equivalence is found, we can continue with the next inequality. Otherwise, we have to find every congruence between the non occurring instance and the occurring instances. As we have already found congruence between a non occurring instance and a representative for it, we can use this representative to find every other congruence for the non occurring instance. We temporarily add implicit edges between a node for the non occurring instance and every node that has an implicit edge with the node of the representative, as we already calculated every congruence for the representative. Now, we can explain the equality, as usual, with the *equality graph* and, after that, remove the temporarily added edges.

Because the `Check` method is always called after the SAT solver has finished with a decision level, only constraints added on this decision level could lead to a conflict. This means that probably a small set of constraints lead to all arising conflicts. Finding enough infeasible subsets so that the SAT solver has to reverse all constraints from this decision level that lead to conflicts, is usually better than calculating and returning infeasible subsets for all conflicts. Calculating an infeasible subset for every conflict, can be expensive and slows down the SAT solver, as it has a lot more clauses to check. This is why we limit the number of returned infeasible subsets.

3.3.3 Removing Constraints, Backtracking

In this section, we present the procedure `Remove` that is invoked when the SAT solver wants to remove a previously added constraint from the theory solver. Algorithm 7 removes one constraint at a time, in the reverse order they were previously added.

Algorithm 7 Removes a constraint

```
 1: function REMOVE(Constraint)
 2:     if IsEquality(Constraint) then
 3:         Stage ← AddedEqualities.Size()
 4:         Lht ← Constraint.LeftHandTerm
 5:         Rht ← Constraint.RightHandTerm
 6:         AddedEqualities.Pop()
 7:         for all t in {Lht,Rht} do
 8:             if IsUF(t) then
 9:                 UfCounter.Decrease(t)
10:         EqualityGraph.RemoveEdges(Stage)
11:         ComputationGraph.Reverse(Stage)
12:     else
13:         AddedInequalities.Pop()
```

Again, we first distinguish between equalities and inequalities (Line 2). Inequalities are just popped from the stack *AddedInequalities* (Line 13). The same thing happens with equalities, they are popped from the stack *AddedEqualities* (Line 6). Just before doing that, we get the *Stage* from the equality we are removing to use it in backtracking (Line 3). We check for both the left hand and right hand term of the equality whether they are uninterpreted function instances. If so, we decrease the according counter by one (Lines 7-9). Then, we remove all implicit and explicit edges in the *equality graph* that are labeled with *Stage* and reverse the *computation graph* (Line 10-11). As explained in Section 3.2.2, when we reverse the *computation graph*, it automatically reverses the integrated *union-find* structure to the state before the removed equality was added.

Occasionally SMT-RAT does not remove the constraint in the reverse order they were added. This is possible, because the SAT solver backtracks and completes a decision level before notifying the theory solver about any changes. It may happen that the SAT solver makes almost the same assignments as before with the exception of on assignment. In this case, SMT-RAT just notifies the theory solver about this change, without making redundant `Remove` and `Add` calls for the other assignments. Another factor that might play into this is, that SMT-RAT is organized in a tree like structure of modules and the theory solver is just one such module with the SAT solver as parent. Other modules might be added with the SAT solver as parent and might influence the behavior of the SAT solver.

If this happens, we do the following: We still just remove inequalities from their stack, but have to expect higher processing cost. When an equality is removed, but is not on top of the stack, we have to remove every equality above it first, otherwise the *computation graph* cannot be kept consistent. We remember the equalities that are removed from the top of the actual removed equality on a stack *ToReassert* as well as the edges that were labeled with their corresponding *Stage*. If the `Remove` method is called with an equality that has already been removed and added to *ToReassert*, we just remove it from that stack.

In the next call of the `Add` or the `Check` method, the first thing we do, is add these equalities from *ToReassert* again in the reverse order they were removed. When we calculate the congruences, we can rely on previous calculations. We can look up all implicit edges that were previously calculated for such an equality and test, whether their representing congruence still holds. As we have not added completely new equalities between the removing and re-adding of the equality, no new congruence can have formed. The removing of an older equality might only have invalidated some of the congruence.

3.4 Heuristics

Now, we have a closer look at how we generate infeasible subsets. There are at least three properties we intuitively prefer in an infeasible subset.

Minimal: An infeasible subset is called minimal, if no proper subset exists that is also an infeasible subset. Thus, a minimal infeasible subset does not contain any redundant constraints.

Small cardinality: The smaller the cardinality of an infeasible subset is, the more it reduces the boolean search space for the SAT solver.

Old: An infeasible subset containing old constraints that were assigned early in the SAT solving process enables the SAT solver to backtrack further back.

An infeasible subset, that fulfills any of the above mentioned properties, may help to speed up the solving process but this is not guaranteed. By using heuristics to generate infeasible subsets that come near to fulfill at least one of these properties, we try to find out whether any of these properties are really desirable in an infeasible subset. Fulfilling both the second and third property is not always possible, as the smallest infeasible subset may contain very young constraints whereas the infeasible subsets only containing the possibly oldest constraints may be very large. In the following, we will present two approaches to explain the equality of two terms in the *equality graph* for generating an infeasible subset.

Approach with Dijkstra's algorithm In this approach, we assign a cost to every edge in the *equality graph* depending on its *Stage* label and whether it is implicit or explicit. We then use Dijkstra's algorithm to find the shortest path between the nodes representing the two terms whose equality we want to explain.

We calculate the cost assigned to an edge as follows: Every edge begins with a cost of 1. We add two different penalties, one for being an implicit edge and one for being a young edge, representing a young equality. The penalty for implicit edges is just a constant cost of 10. The penalty for the age of an edge is calculated through its *Stage* label. We calculate the relative age of the edge by dividing its *Stage* label by the *Stage* of the equality on top of the stack of equalities. This gives us a value between 0 and 1. Edges with an age value near 0 represent equalities that were added very early in the solving process and thereby are very old. Edges with age values near 1 are the opposite and very young. We can choose to punish young edges much more than old edges by exponentiating this age value. Then, we multiply this value with a constant cost of 10 and receive the penalty for young edges. These two penalties are now combined through a weighted mean. The weighted mean allows us to choose, which penalty we want to have more influence on the cost of an edge and thereby on the path Dijkstra's algorithm finds. The formula for calculating the cost of an edge is

$$cost = 1 + \texttt{WeightedMean}(i \cdot 10, \left(\frac{s_{edge}}{s_{current}}\right)^e \cdot 10),$$

where i is a binary value, indicating whether the edge is implicit or not, s_{edge} is the *Stage* of the edges, $s_{current}$ is the *Stage* of the equality on top of the stack and e is the exponent for the age value.

If we place more importance on small infeasible subsets, we might shift the weighted mean to the penalty for implicit edges. This way, Dijkstra's algorithm will more likely choose a path with few implicit edges. This can lead to infeasible subsets with a small cardinality, because we know that we only have to add one equality to the infeasible subset for every explicit edge. For every congruence represented by an implicit edge, we have to add the whole explanation for the congruence to the infeasible subset, which is at least one equality. We could calculate the number of added equalities for every implicit edge and use that number as assigned cost, but this would be very expensive, especially as the *equality graph* is very dynamic and previously calculated numbers might not be accurate anymore.

If we place more importance on using old equalities to explain an equality between two terms, we might shift the weighted mean to the penalty for young edges. This way, Dijkstra's algorithm will more likely choose a path with older edges.

Example 3.4.1 (Approach with Dijkstra)**.** *Let* $\{a = F(x), F(y) = c, x = y, a = b, b = c\}$ *be equalities that were added in this order. Figure 3.16 shows the corresponding* equality graph *where the edges are labeled with their Stage. The Sub Stage of the implicit edge between* $F(x)$ *and* $F(y)$ *is noted behind the dot after the Stage.*

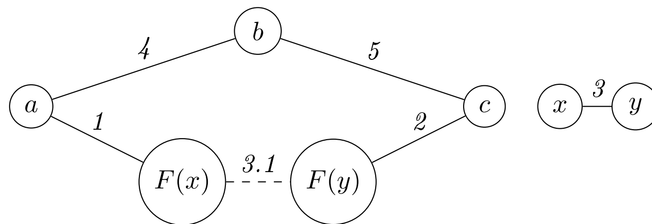

Figure 3.16: *Equality graph* for $\{a = F(x), F(y) = c, x = y, a = b, b = c\}$. Explicit edges are shown as solid edge, implicit edges as dashed edges.

We now want to use the approach with Dijkstra's algorithm to explain the equality between a and c. First, we calculate the cost for every edge. In this example we only consider the penalty for implicit edges. Figure 3.17 shows the corresponding

equality graph *where each edge is labeled with its assigned cost.*

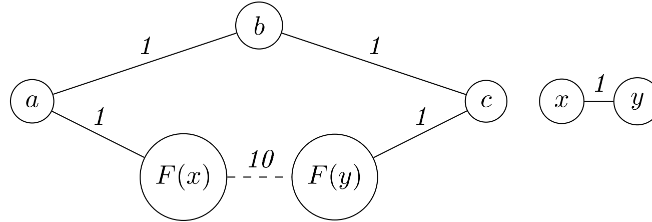

Figure 3.17: *Equality graph* for $\{a = F(x), F(y) = c, x = y, a = b, b = c\}$. Explicit edges are shown as solid edge, implicit edges as dashed edges.

Now we can use Dijkstra's algorithm to find a path from a to c to explain their equality. In this example, Dijkstra chooses the shortest path from a to b to c and returns the explanation $\{a = b, b = c\}$.

With this approach of shifting the importance of small infeasible subsets to old infeasible subsets, we are flexible in choosing the importance we contribute to either of these two properties. It has to be noted, that, with this approach of explaining equality between two terms, we cannot guarantee that a generated infeasible subset is minimal.

Example 3.4.2 (Redundant equalities in explanation). *Let* $\{a = F(a), F(a) = b, b = F(c), F(c) = c\}$ *be the set of added equalities. The equality graph for this set of equalities with cost labels for every edge might look like this:*

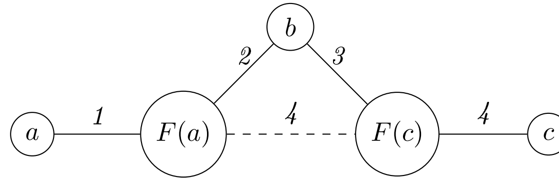

Figure 3.18: *Equality graph* for $\{a = F(a), F(a) = b, b = F(c), F(c) = c\}$. Explicit edges are shown as solid edge, implicit edges as dashed edges.

If we want to generate an infeasible subset that needs an explanation for the equality of $F(a)$ *and* $F(c)$*, Dijkstra's algorithm would just use the implicit edge between the nodes representing these terms. Because this edge represents a congruence, we need to find an explanation for the equality of a and c. Because we are not allowed to use the implicit edge which congruence we are explaining, Dijkstra's algorithm will generate the explanation* $\{a = F(a), F(a) = b, b = F(c), F(c) = c\}$. *This explanation also includes an explanation* $\{F(a) = b, b = F(c)\}$ *for the equality of* $F(a)$ *and* $F(c)$*. The explanation includes redundant equalities and therefor the infeasible subset generated from this explanation is not minimal. The reason that the explanation contains redundancies is, that the graph formed by all found paths is not acyclic.*

Approach with Kruskal In this second approach, we use the same approach as Kruskal's algorithm for finding a minimum spanning tree, to find a tree that connects the nodes of the terms whose equality we want to explain with only the oldest edges. We sort all edges of the according connected component by their *Stage* and *Sub Stage* label and use a new *union-find* structure to track connected components. Initially, every node is in its own set. We process the sorted edges, beginning with the edge with the lowest *Stage*. For each edge, we check whether the nodes it is connecting already are in the same set. If this is not the case, we add the edge to a set *TreeSet* and unify the sets of the two nodes. This is done until the nodes representing the terms of the equality we want to have an explanation for are in the same set. Now, we can use a breadth first search for finding a path

from one node to the other, only using the edges from the set *TreeSet*. This way, we are guaranteed to only use the oldest edges and, therefore, the oldest equalities to explain the equality between the terms.

Explanations generated by this approach never contain redundant equalities and infeasible subsets generated with this approach are always minimal. This approach always finds acyclic paths between two nodes, containing the oldest edges that connect these nodes. If the path only contains explicit edges, the explanation cannot contain any redundant equalities. If an implicit edges is used in a path to explain the equality of two terms a and b, it is used because it is the oldest edge that connects the two components containing a and b. The oldest explanation for the congruence this implicit edges represents has to be slightly older than the implicit edge itself. It can only contain equalities that are older than the implicit edge. Because the implicit edge is the oldest edge that connects the components of a and b, no equality in the explanation of the congruence can also connect these components. This also holds for every other edge used in a path. The resulting graph formed from all found paths in the *equality graph* is always an acyclic graph. Therefore, no equality is explained multiple times and explanations do not contain any redundant equalities.

Example 3.4.3 (Approach with Kruskal)**.** *Let* $\{a = F(x), F(y) = c, x = y, a = b, b = c\}$ *be equalities that were added in this order. Figure 3.19 shows the corresponding* equality graph *where the edges are labeled with their* Stage*. The* Sub Stage *of the implicit edge between* $F(x)$ *and* $F(y)$ *is noted behind the dot after the* Stage*.*

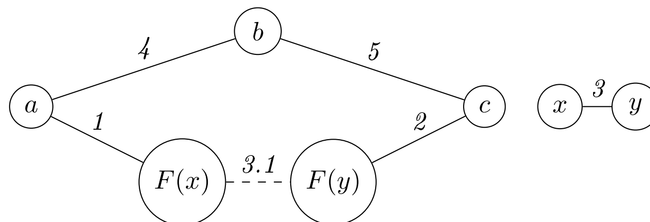

Figure 3.19: *Equality graph* for $\{a = F(x), F(y) = c, x = y, a = b, b = c\}$. Explicit edges are shown as solid edge, implicit edges as dashed edges.

We now want to use the approach with Kruskal's algorithm to explain the equality between a *and* c*. First, we use Kruskal's algorithm to find a tree with only the oldest edges i.e. the edges with the lowest* Stage*. Figure 3.20 shows this tree for the connected component of* a *and* c*.*

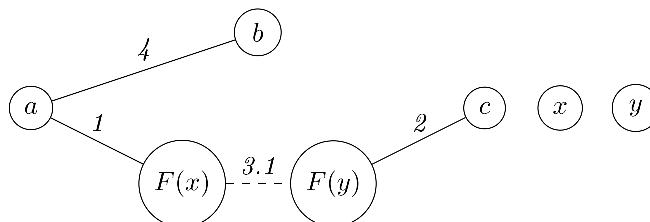

Figure 3.20: Tree found by Kruskal's algorithm

With a breadth-first-search we can find the only path from a *to* $F(x)$ *to* $F(y)$ *to* c *between* a *and* c*. Because this path contains an implicit edge between* $F(x)$ *and* $F(y)$ *we explain the equality between* x *and* y *with the same steps. With this approach we return the explanation* $\{a = F(x), x = y, F(y) = c\}$*.*

3.5 Theory Propagation

As already mentioned, the theory solver is also allowed to return lemmas to the SAT solver. These lemmas are tautologies the theory solver derives from the current set of constraints it has to check. They are used to propagate theory knowledge to the SAT solver with the goal of speeding up the solving process. This is usually done by deducing lemmas that are appended to the formula as unit clauses. This way, we force the SAT solver to directly use the lemma to assign further literals.

Example 3.5.1 (Simple Lemma). *Let $l_1 : a = b$ and $l_2 : b = c$ be assigned constraints. As the theory solver has knowledge about the underlying theory as opposed to the SAT solver, it can make the deduction $d : (a = b \wedge b = c) \Rightarrow (a = c)$. Appending this deduction to the original formula, now forces the SAT solver also to assign the literal for $a = c$, as the literals for $a = b$ and $b = c$ are already assigned.*

Usually, we want to force the SAT solver to assign literals that are already occurring in the original formula. So, one kind of lemma can be deduced just by iterating over all constraints which literals have not yet been assigned. If the left hand term and the right hand term of an equality or inequality are in the same equivalence class, we can generate an explanation from the *equality graph* for this equality. With the equalities in the explanation, we can generate an implication for this equality and return it as lemma to the SAT solver. Now, the SAT solver is forced to assign the literal for this equality to *True*. In the following, we will call this lemma *transitivity lemma*.

Example 3.5.2 (Transitivity Lemma). *Let $\{a = b, b = c\}$ be added constraints with $\{a, b, c\}$ as their equivalence class and $a = c$ a constraint that occurs in the formula we want to solve but whose literal is not yet assigned. When testing the terms of this constraint $a = c$ for equivalence, we find that a and c are in the same equivalence class. Their equality can be explained with the set of constraints $\{a = b, b = c\}$. We can now append the formula $(a = b \wedge b = c) \Rightarrow (a = c)$ and have deducted a* transitivity lemma.

Another type of lemmas can be derived from already added inequalities. While iterating over the added inequalities to find a conflict, we remember inequalities that do not conflict with the equalities. We map the pair of representatives of the equivalence classes in which the terms of the inequality are in to this inequality. Then, we can again iterate over all constraints whose literals are not yet assigned. For each of these constraints we get the pair of representatives for the left hand and right hand term and compare it to the remembered pairs of representatives from the added inequalities. If we find a matching pair of representatives, we can deduce that the left hand and right hand term of the constraint cannot be equal, as there is an added inequality that separates their equivalence classes. For both terms of this constraint, we can generate an explanation for an equality to either of the terms of the added inequality. These explanations, together with the added inequality, implicate the inequality of the terms of the constraint. Now, the SAT solver is forced to assign the literal for this inequality to *True*. In the following, we will call this lemma *inequality propagation lemma*.

Example 3.5.3 (Inequality Propagation Lemma). *Let $\{a = b, c = d, a \neq c\}$ be added constraints with $\{a, b\}, \{c, d\}$ as their equivalence classes and $b = d$ a constraint that occurs in the formula we want to solve but whose literal is not yet assigned. When testing the terms of this constraint $b = d$ for equivalence, we find that b and d are in different equivalence classes. Furthermore, we see that there is an added inequality $a \neq c$ separating their equivalence classes. With explanations for the equality of a and b, the equality of c and d and the assigned inequality*

$a \neq c$, we can deduce the inequality $b \neq d$. We combine the respective explanations $\{a = b\} \cup \{c = d\} \cup \{a \neq c\}$ to explain the inequality $b \neq d$ in a lemma $(a = b \wedge c = d) \wedge a \neq c \Rightarrow (b \neq d)$. Now, this lemma is appended to the formula.

In some cases, it might also be beneficial to deduce lemmas that implicate a new constraint not yet occurring in the original formula. These lemmas are called constructive lemmas. One deduction we could return as constructive lemma is the explanation for a congruence between two uninterpreted function instances. We can just add an implication from this explanation to the equality of the uninterpreted function instances. As we often already have to explain congruence while explaining the equality of two terms, this lemma comes cheap. In the following, we will call this lemma *congruence lemma.*

Example 3.5.4 (Congruence Lemma)**.** *Let $\{a = b, F(a) = c, F(b) = d\}$ be added constraints. Some operation wants to have the equality between c and d explained. This equality only holds because of the congruence between $F(a)$ and $F(b)$. While generating this explanation, we have to explain this congruence with $\{a = b\}$. This explanation now can be used to create a* congruence lemma $(a = b) \Rightarrow (F(a) = F(b))$ *that is appended to the formula. It is possible, that $F(a) = F(b)$ is an equality that does not occur in the original formula. Therefore, we have added a new equality to the formula and the lemma is a constructive lemma.*

As with infeasible subsets, a benefit from generating lemmas is not guaranteed. If we would deduce all possible lemmas at once, we would transform this lazy SMT solving approach to an eager SMT solving approach, which the SAT solver is not necessarily optimized for. Lemmas are usually deduced from currently assigned literals. This means, they can only advance the SAT solving process as long as the literals used in the lemma are assigned to *True*. For example, this means we should not return congruence lemmas while generating an infeasible subset to explain a conflict. Due to the infeasible subset, the SAT solver will backtrack and reverse the assignment of some literals, very likely one used in the lemma. From that moment on, the lemma is not used to propagate assignments and might never be used again in the rest of the solving process. Therefore, we should return lemmas only when no conflict is found and the lemmas are actually used to propagate the assignment of more literals.

We only generate lemmas on every $n-$th `Check` call that does not find a conflict and also limit the number of lemmas generated per such a `Check` call.

Chapter 4

Conclusion

The goal of this thesis was to present a theory solver for the theory of equality logic with uninterpreted functions which is very flexible in ways to generate infeasible subsets.

4.1 Summary

First, we established a general background knowledge on the SAT and SMT problem and popular approaches to solve them. Then, we presented the theory of equality logic with uninterpreted functions and what calculations are needed to check a set of constraints from this logic for consistency. Next, we moved on to the actual theory solver by explaining the used data structures and procedures it provides. We presented two heuristic for generating infeasible subsets through the *equality graph*. One is based on Dijkstra's algorithm for finding the shortest path and has several parameters for generating infeasible subsets with different properties, the other one is based on Kruskal's algorithm for finding a minimal spanning tree to find the oldest infeasible subset. Additionally, we presented three kinds of lemmas we can generate in an attempt to speed up the solving process. The presented theory solver fulfills the three named requirements for being integrated in lazy SMT solving: it is *incremental*, as it is able to receive equalities and inequalities at any time, can generate *infeasible subset* in various ways and is able to *backtrack* when equalities and inequalities are removed.

4.2 Experimental Results

In the following, we will show some experimental result. We applied our SMT solver on the QG-classification benchmark set provided by [SMT] that consists of 5262 examples to test and compare the different heuristics of our theory solver with various settings. For each EUF formula, the solver has 30 seconds to decide whether it is satisfiable or not. The following graphs all show the percentage of formulas that could be solved within 30 seconds. The graphs only show the percentage of formulas solved above 50%, otherwise, the differences between the various results would be difficult to distinguish.

First, we wanted to find out, what limit l of generated infeasible subsets per `Check` call works best for both heuristics. In Figure 4.1 we see the results for the approach with Kruskal's algorithm and various limits for the generated infeasible subsets. It solves most EUF formulas within 30 seconds with a limit of 5 infeasible subsets per `Check` call. We see worse performance with higher and lower limits than 5.

In Figure 4.2 we see the results for the approach with Dijkstra's algorithm and various limits for the generated infeasible subsets. We fixed the settings for

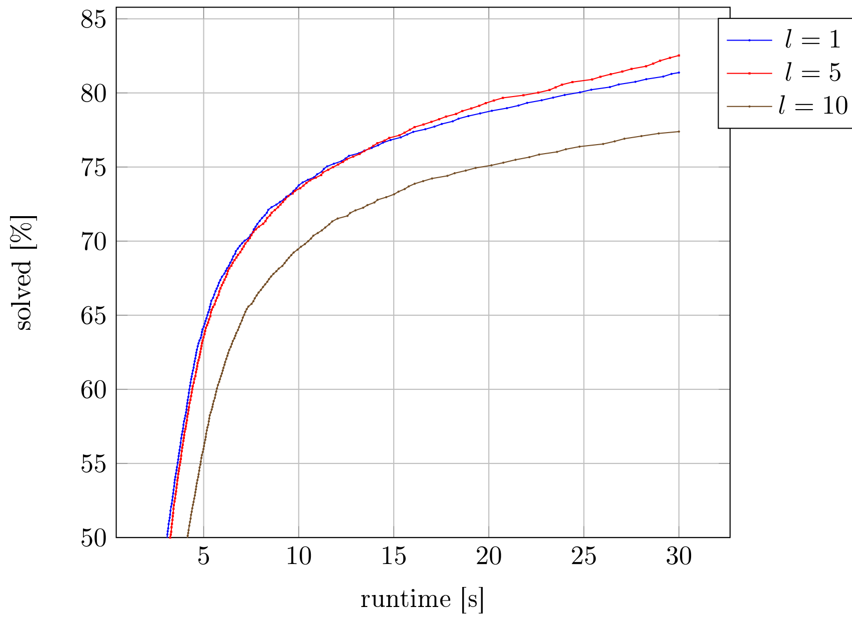

Figure 4.1: Varying limit on generated infeasible subsets per `Check`, Kruskal

this heuristic to calculate the arithmetic mean between the penalties and used an exponent of 1 for the penalty for young edges. It solves most EUF formulas within 30 seconds with a limit of 1 infeasible subset per `Check` call. Right from the beginning, the results differ slightly for each limit. The higher the limit of infeasible subsets this approach is allowed to generate per `Check` call, the worse are its results.

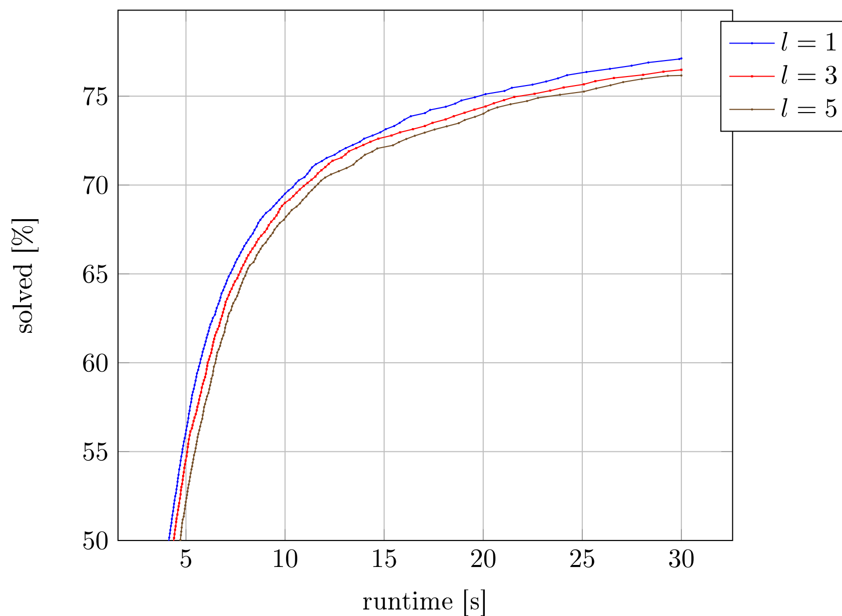

Figure 4.2: Varying limit on generated infeasible subsets per `Check`, Dijkstra

Next, we compared different settings for heuristic with Dijkstra's algorithm. We fixed the limit of infeasible subsets to one and varied the weighted mean, shifting the weight more to either penalty. The variable w is the factor for the penalty for implicit edges and $(1 - w)$ is the factor for the penalty for young edges. The

closer w is to 1, the more we shift to the penalty for implicit edges, the closer it is to 0, the more we shift to the penalty for young edges. As we can see in Figure 4.3, the results do not differ much for a w between 0 and 0.5. For w higher than 0.5, the performance worsens drastically. Using Dijkstra's algorithm to find older explanations seems to work better than finding smaller explanations.

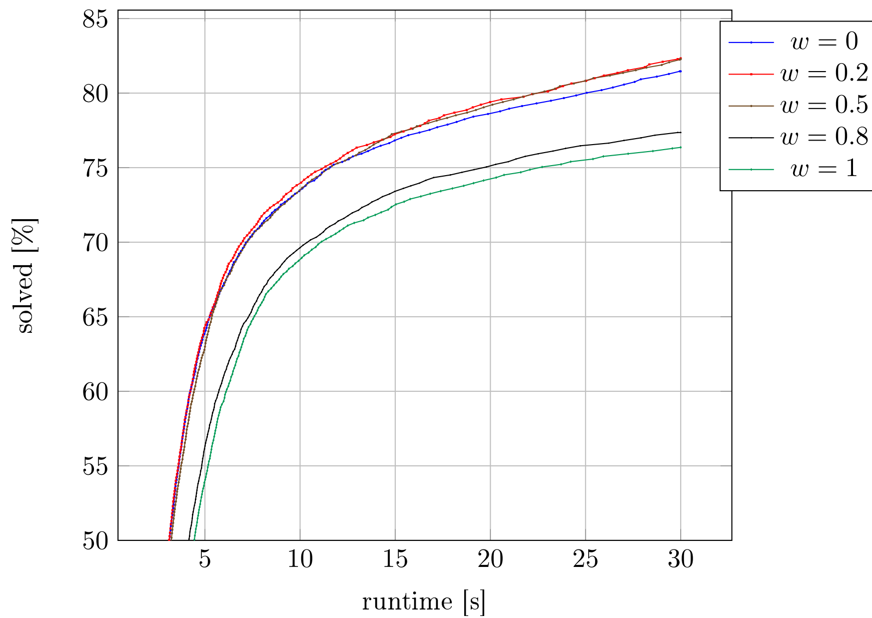

Figure 4.3: Comparing different settings for Dijkstra

In Figure 4.4, we compare the best settings for both heuristics with a version of the same theory solver that does not generate any infeasible subset. We see that both heuristics perform very similar and slightly better, especially on bigger problems.

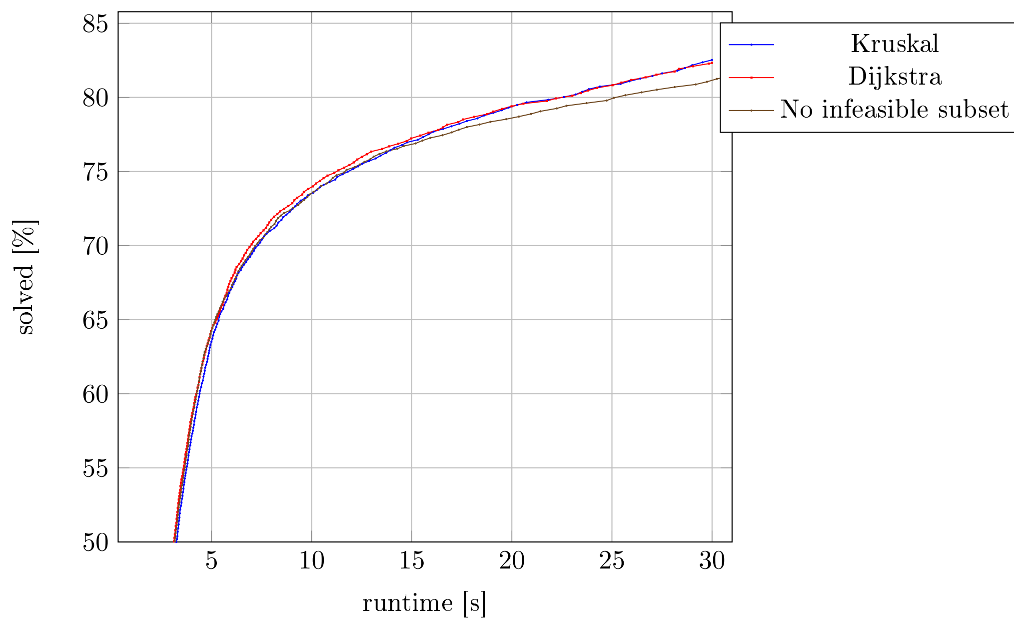

Figure 4.4: Comparing Kruskal and Dijkstra

Then, we compared the effects of generating different kinds of lemmas on the solving process. For both approaches, we compared the performance of the heuristic without generating lemmas (N), only with generating either *transitivity lemmas* (T) or *inequality propagation lemmas* (I), *transitivity lemmas* and *inequality propagation lemmas* together and at last both lemmas combined with the *congruence lemma* (C).

In Figure 4.5 we see the results for the approach with Kruskal. Generating lemmas decreased its performance, except for the *inequality propagation lemma*. Especially the *congruence lemma* slowed down the solver so much, that we could not include it in this figure. We suspect that this slowdown is due to the amount of lemmas that are generated. Further tests have to be conducted to confirm this.

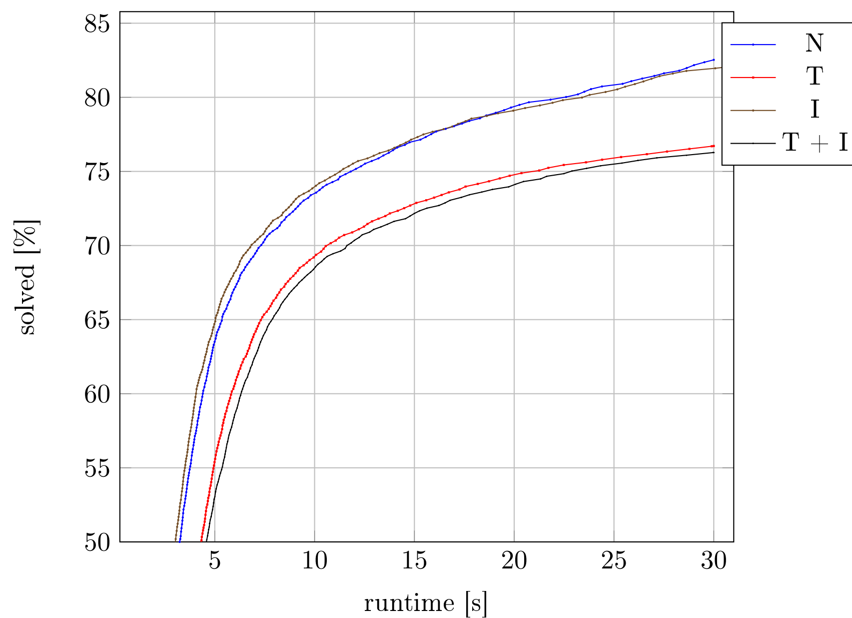

Figure 4.5: Kruskal with lemmas

In Figure 4.6 we see the results for the approach with Dijkstra. Both, the *transitivity lemma* and the *inequality propagation lemma* alone worsened the performance of the solver. But, when they were combined, they solved smaller problems significantly faster than without lemmas. Adding *congruence lemmas* did not seem to have much of an effect.

Especially Figure 4.6 shows that the performance of the solver depends a lot on the specific settings we choose. The theory solver we presented has a lot of parameters and finding good parameters is crucial for the performance.

At last, we compared the best performances of both heuristics, including lemmas, with the theory solver that does not generate infeasible subsets in Figure 4.7. The approach with Kruskal improves the performance on bigger problems but shows no difference for smaller problems. The approach with Dijkstra improves the performance on smaller problems considerably but looses performance on bigger problems.

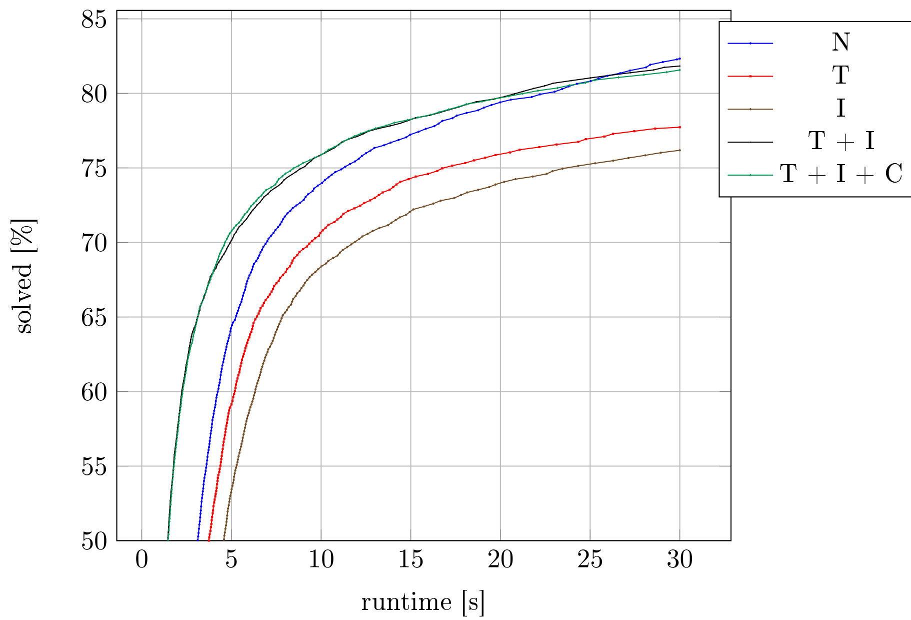

Figure 4.6: Dijkstra with lemmas

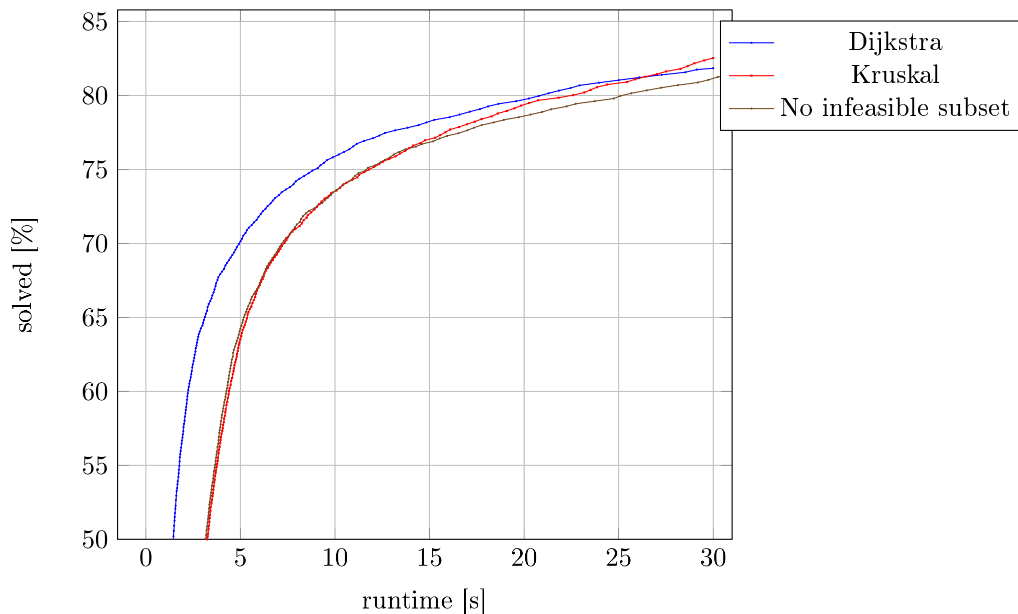

Figure 4.7: Best performances of Kruskal and Dijkstra

4.3 Future work

The presented theory solver has lots of options for optimizations. First, the currently implemented heuristics and lemmas have many settings that strongly influence the performance. As it is difficult to find the right settings to achieve a better performance, more tests with different settings have to be conducted. The *equality graph* is a very general data structure that allows many other heuristics to generate infeasible subsets and lemmas with various graph algorithms. As we have seen so far, heuristics that try to find older infeasible subsets seem to perform better in the solving process. An alternative to the *equality graph* could be implemented as directed acyclic graph. This approach is presented in [NO05]and allows a very fast generation of explanations for equalities, but is limited to the

oldest explanations only. This paper also presents an elegant method to incrementally calculate congruence closure but requires some preprocessing on the original formula. There are also many other ways how we can try to speed up the solving process with preprocessing. For example, we could perform eager SMT solving approaches, named in the introduction, on small subsets of the original formula. Symmetries in the original formula can be exploited as described in [DFMP11].

Bibliography

[Ack54] Wilhelm Ackermann. Solvable Cases of the Decision Problem. In *Studies in Logic and the Foundations of Mathematics*. 1954.

[BD94] Jerry R. Burch and David L. Dill. Automatic Verification of Pipelined Microprocessor Control. In David L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer Berlin Heidelberg, 1994.

[BGV99] Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 470–482. Springer Berlin Heidelberg, 1999.

[BGV01] Randal E. Bryant, Steven German, and Miroslav N. Velev. Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. *ACM Trans. Comput. Logic*, 2(1):93–134, January 2001.

[BV02] Randal E. Bryant and Miroslav N. Velev. Boolean Satisfiability with Transitivity Constraints. *ACM Trans. Comput. Logic*, 3(4):604–627, October 2002.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Proc. of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *LNCS*, pages 360–368. Springer, September 2015.

[DFMP11] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Automated Deduction–CADE-23*, pages 222–236. Springer, 2011.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[GSZ+03] Anuj Goel, Khurram Sajid, Hai Zhou, Adnan Aziz, and Vigyan Singhal. BDD Based Procedures for a Theory of Equality with Uninterpreted Functions. *Formal Methods in System Design*, 22(3):205–224, 2003.

[HIKB96] Ramin Hojati, Adrian Isles, Desmond Kirkpatrick, and Robert K. Brayton. Verification using uninterpreted functions and finite instantiations. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 218–232. Springer Berlin Heidelberg, 1996.

[HKGB97] Ramin Hojati, Andreas Kuehlmann, Steven German, and Randal K. Brayton. Validity checking in the theory of equality with uinterpreted functions using finite instantiations. In *Unpublished paper presented at the International Workshop on Logic Synthesis*, 1997.

[KS08] Daniel Kröning and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications*, RTA'05, pages 453–468, Berlin, Heidelberg, 2005. Springer-Verlag.

[PSS98] Amir Pnueli, Ofer Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin Heidelberg, 1998.

[Seb07] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.

[Sho78] Robert E. Shostak. An Algorithm for Reasoning About Equality. *Commun. ACM*, 21(7):583–585, July 1978.

[SMT] The Satisfiability Modulo Theories Library. `http://www.cs.nyu.edu/~barrett/smtlib/`.

[Tar75] Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, April 1975.