Implementation of a Fast Congruence Closure Algorithm

Patrick Bahr

Abstract

In this paper an abstract algorithm for computing the congruence closure of a set of ground equations using the standard union-find infrastructure is given as well as an abstract algorithm that decides whether a ground equation is a semantic consequence of a set of ground equations using the output of the congruence closure algorithm. Furthermore an efficient C++ implementation of both algorithms is given using fast *weighted* union and long find with path compression. Additionally we argue that the given congruence closure algorithm runs in $\mathcal{O}(n \log n)$ time for n being the total size of the input equations and that the decision algorithm performs in linear time in the size of the equation that is to be checked, independently of the size of the equations that induced the congruence closure.

1 Introduction

Deciding the word problem for a given finitely presented equational theory, i.e., deciding if an equation follows from a finite set of other equations, is the cornerstone of many verification and deduction systems. The interpretation of equations as a term rewriting system together with the *Knuth-Bendix completion* provides at least a semi-decision procedure. In the particular case that the equational theory is given by ground equations Knuth-Bendix completion always terminates thus yielding a decision procedure. However the complexity of the ground completion is indeed polynomial but then the word problem for a given ground equation is polynomial in the size of the resulting completed set of equations and the size of the equation in question.

The alternative approach to the word problem for ground equations is to compute the *congruence closure* of the given equalities and then decide the equation w.r.t. the computed closure. The standard technique to accomplish this is to use the well-known union-find algorithms and data structures to maintain the equivalence relation induced by the initial equalities and further ones introduced by the algorithm to gain the congruence property. The idea of the union-find equivalence closure algorithm is to present the equivalence relation as its quotient set, i.e. the set of all equivalence classes. union then merges two equivalence classes, i.e. imposes an equality between some respective representatives and find provides the equivalence class of a given element.

What then remains to be established to get a congruence relation is to ensure compatibility of the equation relation $\approx$ with every function symbol of the signature, i.e., for all $k \geq 0$ and all k-ary function symbols f, if we have (i) $t_i \approx s_i$ for all $1 \leq i \leq k$, then also (ii) $f(t_1, \dots, t_k) \approx f(s_1, \dots, s_k)$. The idea is now to check (i) by find and if it holds true to establish (ii) by union. This is done systematically for every subterm occurring in the equations until a fixed point is reached. As only finitely many terms are considered this fixed point is reached after finitely many iterations. Including also other terms that are not subterms of the initially given equations is implicitly done by the decision algorithm.

2 Preliminaries

At first we want to formalise what the word problem for a set of ground equation actually is to see then, that congruence closure really solves this problem.

Definition 1. An $\mathbb{N}$-indexed family $\Sigma = \{\Sigma^{(k)}\}_{k\in\mathbb{N}}$ is called a *signature.* An element $f \in \Sigma^{(k)}$ is called a k-ary function symbol. 0-ary function symbols are also called constants.

Definition 2. Let Σ be a signature. The set *ground* Σ*-terms* of T_Σ is inductively defined by:
If $k \geq 0$, $f \in \Sigma^{(k)}$ and $t_i \in T_\Sigma$ for all $1 \leq i \leq k$, then $f(t_1, \dots, t_k) \in T_\Sigma$.

Definition 3. Let Σ be a signature and $t \in T_\Sigma$. The set $\mathcal{P}os(t)$ of all *positions in* t is recursively defined by:
If $t = f(t_1, \dots, t_k)$ for $k \geq 0$ then $\mathcal{P}os(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(t_i)\}$

Definition 4. Let Σ be a signature and $t \in T_\Sigma$. The *size of* t, denoted $|t|$, is the number of positions in t, i.e. $|t| = |\mathcal{P}os(t)|$.

Definition 5. Let Σ be a signature, $t = f(t_1, \dots, t_k) \in T_\Sigma$ and $p \in \mathcal{P}os(t)$. The *function symbol in term* t *at position* p, denoted $t(p)$, is recursively defined as follows:

$$t(p) = \begin{cases} f & \text{if} \quad p = \varepsilon \\ t_i(p') & \text{if} \quad p = p'i \quad \text{for some } p' \in \mathcal{P}os(t_i) \end{cases}$$

Definition 6. Let Σ be a signature and $\equiv$ an equivalence relation on T_Σ. $\equiv$ is called a *congruence relation* if the following condition holds:
If for some $k \geq 0$ we have $t_i \equiv s_i$ for all $1 \leq i \leq k$ and $f \in \Sigma^{(k)}$ then also $f(t_1, \dots, t_k) \equiv f(s_1, \dots, s_k)$.

Definition 7. Let Σ be a signature and $s, t \in T_\Sigma$. $s \approx t$ is called a *ground* Σ*-equation.*

Definition 8. Let Σ be a signature. A Σ*-algebra* $\mathcal{A}$ is a pair $(A, \cdot^\mathcal{A})$ where

- A is a *non-empty* set, and
- $\cdot^\mathcal{A}$ is a mapping that associates with each k-ary function symbol $f \in \Sigma^{(k)}$ a function $f^\mathcal{A} : A^k \to A$ for all $k \geq 0$.

Definition 9. Let Σ be a signature and $\mathcal{A}$ and $\mathcal{B}$ Σ-algebras. A mapping $\phi : A \to B$ is called Σ*-homomorphism* from $\mathcal{A}$ to $\mathcal{B}$, denoted $\phi : \mathcal{A} \to \mathcal{B}$, iff for all $k \geq 0$, $f \in \Sigma^{(k)}$, and $a_1, \dots, a_k \in A$ we have $\phi(f^\mathcal{A}(a_1, \dots, a_k)) = f^\mathcal{B}(\phi(a_1), \dots, \phi(a_k))$.

Note that $\mathcal{T}_\Sigma = (T_\Sigma, \cdot^{\mathcal{T}_\Sigma})$, where $f^{\mathcal{T}_\Sigma}(t_1, \dots, t_k) = f(t_1, \dots, t_k)$, is a Σ-algebra such that for every Σ-algebra $\mathcal{A}$ there is exactly one homomorphism $\phi : \mathcal{T}_\Sigma \to \mathcal{A}$, i.e., $\mathcal{T}_\Sigma$ is said to be initial in the class of all Σ-algebras.

Definition 10. Let Σ be a signature, $s \approx t$ a ground Σ-equation, E a set of ground Σ-equations, and $\mathcal{A}$ a Σ-algebra.

(i) $s \approx t$ *holds* in $\mathcal{A}$, denoted $\mathcal{A} \models s \approx t$, iff for the unique homomorphism $\phi : \mathcal{T}_\Sigma \to \mathcal{A}$ we have $\phi(s) = \phi(t)$.

(ii) $\mathcal{A}$ *models* E, denoted $\mathcal{A} \models E$, iff every equation in E holds in $\mathcal{A}$.

(iii) $s \approx t$ is *semantic consequence* of E, denoted $E \models s \approx t$, iff for all Σ-algebras $\mathcal{B}$ we have that $\mathcal{B} \models E$ implies $\mathcal{B} \models s \approx t$.

(iv) The relation $\approx_E = \{(s,t) \in T_\Sigma \times T_\Sigma | E \models s \approx t\}$ is called the *equational theory of E*.

The word problem for a given set of ground Σ-equations and two ground Σ-terms s, t is now the question whether $s \approx_E t$ holds. That considering the congruence closure is the solution for this problem is stated in the following theorem.

Theorem 11. *Let Σ be a signature and E be a set of ground Σ-equations. $\approx_E$ is the smallest congruence relation containing E.*

This is basically an immediate consequence of Birkhoff's Theorem (cf. [2, lemma 3.5.12, Theorem 3.5.14]).

3 Congruence Closure and Decision Algorithm

The algorithm that is presented in the following is based on the algorithm in [3].

For this to work each term has to be interpreted as a graph having a vertex for each position in the term and edges from each position to its direct "subposition".

Definition 12. Let Σ be a signature, $t \in T_\Sigma$ and E a set of ground Σ-equations.

(i) The labelled digraph $G_t = (V_t, E_t, l_t)$, where $V_t = \mathcal{P}os(t)$, $l(p) = t(p)$ for all $p \in V_t$, $E_t = \{(p, p') \in V_t^2 \,|\, \exists i \in \mathbb{N}.p' = pi\}$ and for each node $p \in V_t$ the set of successors of p is ordered by $p1 < \dots < pk$, is called the graph of t. Note that G_t is a tree. By $v_t \in V_t$ we denote the root of this tree.

(ii) The graph $G_E = \biguplus_{s \approx s'} G_s \uplus G_{s'}$ is called the graph of E.

The input of the algorithm will be G_E, the graph of a finite set E of ground Σ-equations for a fixed signature Σ. In the following we will only speak of a finite labelled graph $G = (V, E, l)$ for which every node's successors are ordered, oblivious of the fact that the graph originated from a finite set of ground Σ-equations. As already mentioned the algorithm will use the common union-find infrastructure to maintain the properties of an equivalence relation. For this we need a set C of names for the equivalence classes. This set must have at least as many elements as V. For convenience we assume C to be countably infinite and that $V \subseteq C$. Then we have the following operations on the set of equivalence relation representations $Eq_{G,C}$[1] on the graph G with equivalence class names from C:

- find : $Eq_{G,C} \times V \to C$;
  where $(R, v) \mapsto$ name of the equivalence class in R containing the node v.
- union : $Eq_{G,C} \times C \times C \to Eq_{G,C}$;
  where $(R, c_1, c_2) \mapsto R'$, and R' is the representation where the equivalence classes named c_1 and c_2 are combined into an equivalence class named c_1.

[1] This can be seen as the set C^V of all mappings from vertices of G to equivalence class names in C.

- $\mathsf{pred} : Eq_{G,C} \times C \to 2^V$;
  where $(R, c) \mapsto \{v \in V \mid \exists v'.\mathsf{find}(R, v') = c \text{ and } (v, v') \in E\}$
  I.e., (R, c) is mapped to the set of vertices having a successor in the equivalence class named c.

- $\mathsf{weight} : Eq_{G,C} \times C \to \mathbb{N}$;
  where$(R, c) \mapsto |\{v \in V \mid \mathsf{find}(R, v) = c\}|$
  I.e., (R, c) is mapped to the number of elements that are in the equivalence class named c.

To establish the congruence closure we need another data structure that keeps track of the equivalence classes of the successors of nodes. Therefore, we need the notion of a signature table which captures this idea.

Definition 13. Let Σ be a signature, $G = (V, E, l : V \to \Sigma)$ a labelled digraph, v a node in G, $v_1 < \cdots < v_k$ its successors and $R \in Eq_{G,C}$. The Σ, C*-signature* of v w.r.t. R is the $(k + 1)$-tuple $sig(R, v) = (l(v), \mathsf{find}(R, v_1), \ldots, \mathsf{find}(R, v_k))$. The set of all Σ, C-signatures is denoted by $Sig_{\Sigma,C} := \Sigma \times C^*$. A Σ, C-signature table is a partial mapping $\tau : Sig_{\Sigma,C} \rightharpoonup C$ that maps a Σ, C-signature to an equivalence class name. The empty Σ, C-signature table is denoted by ε, s.t. $\varepsilon(s) = \perp$ for all $s \in Sig_{\Sigma,C}$. The domain of a Σ, C-signature table τ is the set $dom(\tau) = \{s \in Sig_{\Sigma,C} \mid \tau(s) \neq \perp\}$.

Additionally we need some operation to work on a Σ, C-signature table τ for some $s' \in Sig_{\Sigma,C}$ and $c \in C$:

- $\tau' = \tau[s' \mapsto c]$ is defined as $\tau'(s) = \begin{cases} c & \text{if } s = s' \\ \tau(s) & \text{otherwise} \end{cases}$ for all $s \in Sig_{\Sigma,C}$

- $\tau' = \tau_{\setminus s'}$ is defined as $\tau'(s) = \begin{cases} \perp & \text{if } s = s' \\ \tau(s) & \text{otherwise} \end{cases}$ for all $s \in Sig_{\Sigma,C}$

So $\tau[s' \mapsto c]$ sets the image of s' to c and $\tau_{\setminus s'}$ removes s' from the domain of τ.

Now we can give the algorithm that computes the congruence closure for the Graph G_E of a given finite set E of ground Σ-equations. The algorithm is given in figure 1. It starts by initialising the equivalence relation representation as identifying those nodes v_s, v_t that correspond to terms s, t for which we have an equation $s \approx t \in E$. The Σ, C-signature table τ is initialised as empty table. Furthermore we have two sets *pending* and *combine*. The set *pending* contains those nodes v that need to be checked for their signature. If v's signature is also found in the Σ, C-signature table, say being mapping to the equivalence class c', the pair (c, c') is put into the set *combine*, for c being the equivalence class of v, provided c and c' are not equal. This set *combine* contains pairs of eqivalence class (names) that both contain a respective node that have been found to be equivalent (by having the same Σ, C-signature). Thus, these two equivalence classes have to be merged using union. For the sake of efficiency the smaller class, say c, is merged into the bigger one, say c' — hence, weighted union. Since the class c is now c' all nodes that had a Σ, C-signature containing c are removed from the Σ, C-signature table and put into *pending* for further investigation. This process is iterated until a fixed point is reached, i.e., *pending* is empty. The result of the computation is then the Σ, C-signature table τ which contains for a signature s the equivalence class (name) $\tau(s)$ that contains all nodes having the signature s.

The algorithm given in figure 2 that decides a given equation $s \approx t$ uses as input the Σ, C-signature table τ that was produced by the congruence closure algorithm. The decision algorithm now simply computes recursively the respective equivalence class of both terms using the Σ, C-signature table τ. If the algorithm encounters a Σ, C-signature s that is not in the domain of τ, a fresh equivalence class name c from C is taken and τ is changed to map s to c.

```
input: set $E$ of ground $\Sigma$-equations
$\tau \leftarrow \varepsilon$    ▷ Initialise $\Sigma, C$-signature table $\tau$ as empty
set $R$ s.t. $\mathsf{find}(R, v) = v$ for all $v \in V_E$.
for $s \approx t \in E$ do    ▷ Impose the desired equalities.
    $R \leftarrow \mathsf{union}(R, v_s, v_t)$
end for
$\mathit{pending} \leftarrow V_E$
while $\mathit{pending} \neq \emptyset$ do
    $\mathit{combine} \leftarrow \emptyset$
    for $v \in \mathit{pending}$ do
        if $\tau(\mathit{sig}(v)) = \perp$ then
            $\tau \leftarrow \tau[\mathit{sig}(v) \mapsto \mathsf{find}(R, v)]$
        else if $\mathsf{find}(R, v) \neq \tau(\mathit{sig}(v))$ then
            $\mathit{combine} \leftarrow \mathit{combine} \cup \{(\mathsf{find}(R, v), \tau(\mathit{sig}(v)))\}$
        end if
    end for
    $\mathit{pending} \leftarrow \emptyset$
    for $(e_{1,2}) \in \mathit{combine}$ do
        if both $e_1$ and $e_2$ are used in $R$ then    ▷ i.e. $\mathsf{find}(R, v_i) = e_i$ for some $v_i$
            if $\mathsf{weight}(R, e_1) < \mathsf{weight}(R, e_2)$ then    ▷ Merge smaller class into bigger one.
                $(e_1, e_2) \leftarrow (e_2, e_1)$    ▷ Swap if $e_1$ is smaller than $e_2$.
            end if
            for $u \in \mathsf{pred}(R, e_2)$ do
                $\tau \leftarrow \tau_{\setminus \mathit{sig}(u)}$
                $\mathit{pending} \leftarrow \mathit{pending} \cup \{u\}$
            end for
            $R \leftarrow \mathsf{union}(R, e_1, e_2)$
        end if
    end for
end while
output: $\tau$
```

Figure 1: Congruence closure algorithm.

```
input: ground $\Sigma$-equation $s \approx t$, $\Sigma, C$-signature table $\tau$
output: true if $\textsc{Class}(s) = \textsc{Class}(t)$; otherwise false
function $\textsc{Class}(t = f(t_1, \ldots t_k) \in T_\Sigma)$
    for $i \in \{1, \ldots k\}$ do
        $c_i = \textsc{Class}(t_i)$
    end for
    if $\tau((f, c_1, \ldots, c_k)) = \perp$ then
        take some $c \in C \setminus \mathit{dom}(\tau)$
        $\tau \leftarrow \tau[(f, c_1, \ldots, c_k) \mapsto c]$.
    else
        $c \leftarrow \tau((f, c_1, \ldots, c_k))$
    end if
    return $c$
end function
```

Figure 2: Decision algorithm.

4 Implementation

For the implementation of the algorithms presented above we chose C++. But before we get into the details, we have to find a C++ implementation for the data structures used in the abstract algorithms.

Most importantly we have to commit ourselves to a particular style of representing equivalence classes. There are two complementary approaches. On the one hand the membership of a node in an equivalence class can be represented as a lookup table that provides the equivalence class for each node. Using this representation find can be implemented to run in constant time whereas union takes in general $\mathcal{O}(n)$ time for $n = |V|$.

On the other hand equivalence classes can be represented as trees such that all nodes of the tree are considered to be in the same equivalence class. Then the root of a tree can be taken as the canonical element representing the equivalence class formed by the tree. union of two equivalence class represented as roots of some trees is then just a matter of making the root of the one tree a child of the other tree's root. This can be done in constant time, whereas find needs to travel up the tree to root and hence can take $\mathcal{O}(n)$ time in the worst case.

For a more detailed presentation see [1]. There it is also shown that the latter approach always outperforms the first one. Therefore, we chose this method to represent equivalence classes.

Now we take a look on how to represent signatures, terms and Σ, C-signatures.

For each symbol we have an object of class Symbol containing the string representation and its arity:

```
class Symbol {
public:
  Symbol(const string name, const int arity);
  virtual ~Symbol();
  const string& getName() const;
  int getArity() const;

  size_t hash() const;
  bool operator==(const Symbol &s) const;
  bool operator<(const Symbol &s) const;

private:
  const string name;
  const int arity;
};
```

The additional operator overridings and the hash() method are necessary to use the hash table implementation provided by the C++ Standard Template Library (STL) which will be necessary as symbols are included in Σ, C-signatures.

The signature Σ is taken to be implicitly given by the input equations. That is, during the parsing a table is created that maps strings to their symbol representation. This table will be of the following hash map type:

```
typedef hash_map<string,Symbol const*> ArityTable;
```

Based on this definition terms can be defined easily:

```
class Term {
public:
  Term(Symbol const* symbol, Term** args);
  Term(Symbol const* symbol, Term** args,Term* merge);
  virtual ~Term();
  Symbol const* getSymbol() const;
  Term**const getArgs() const;
  Term* operator[](int index) const;

protected:
  Symbol const*const symbol;
  Term**const args;
};
```

Note that this is rather the representation of the corresponding graph of a term as two different objects of this type can represent the same term. That is why we will call objects of type Term also nodes.

A Σ, C-signature is similarly represented as a term, with the difference that the arguments of a function symbol are now elements of C. As we will want to use different representations of C, we abstract from the respective type representing C:

```
template<class T> class Signature
{
public:
  Signature(Symbol const* symbol);
  virtual ~Signature();

  size_t hash() const;
  Symbol const* getSymbol() const;
  int getArity() const;
  T& operator[](const int index) const;

  bool operator==(const Signature<T>& s) const;
  bool operator<(const Signature<T>& s) const;

private:
  Symbol const*const symbol;
  T* signature;
};
```

A Σ, C-signature table is then just a hash table having Signature<EqClass> as key type and EqClass as value type for some type EqClass representing names of equivalence classes.

```
typedef hash_map<Signature<EqClass>*,EqClass> SignatureTable;
```

To represent equivalence classes of objects of class Term, we have to extend the class Term to a subclass called CCTerm that provides the structure needed to build trees that represent the equivalence classes. Therefore, each object of class CCTerm has a member equivClass that points to its parent in the equivalence class tree or to itself if it is the root. In addition some members needed for bookkeeping are included. Those do only have a meaningful value for CCTerm objects representing an equivalence class (i.e. those that are the root of an equivalence class tree). weight contains the number of elements in the respective equivalence class, predList is the list representation of pred(R, c) where R is the current equivalence representation and c the equivalence class represented by the CCTerm object. auxMember is used later when normalising the resulting Σ, C-signature table to an equivalent one that uses integers for representing equivalence classes.

```
class CCTerm : public Term {
public:
  CCTerm(Symbol const* symbol, CCTerm** args,CCTerm* merge);
  CCTerm(Symbol const* symbol, CCTerm** args);
  virtual ~CCTerm();

  bool isEqClass() const;
  EqClass getEqClass() ;
  Signature<EqClass>* getSignature() const;

  PredList* getPredList() const;
  CCTerm* operator[](int index) const;
  int getAuxMember() const;
  void setAuxMember(int aux);
  int getWeight() const;
  void unionEqClass(EqClass eq);

private:
  PredList* predList;
  CCTerm* equivClass;
  int auxMember;
  int weight;
};
```

The additional constructor expecting an argument merge is used to do the initial union of the terms in the equation at parse time. If it is NULL it is the same as the constructor without this additional argument:

```
CCTerm::CCTerm(Symbol const* symbol, CCTerm** args,CCTerm* merge)
:Term(symbol,(Term**)args),
predList(merge == NULL ? new PredList() : NULL),
equivClass(merge == NULL ? this : merge->getEqClass()),
auxMember(-1)
,weight(0) // is incremented to 1 later
{
  equivClass->weight++;
}
```

The type PredList is just a STL list of pointers to CCTerm, and EqClass is just an alias for CCTerm∗ that is used if we expect a CCTerm object that represents a equivalence class:

```
typedef list<CCTerm*> PredList;
typedef CCTerm* EqClass;
```

Let us take a closer look on the implementation of some of the methods of the CCTerm class.

Computing the Σ, C-signature of a node is quite simple:

```
Signature<EqClass>* CCTerm::getSignature() const {
  Signature<EqClass>* sig = new Signature<EqClass>(symbol);
  for(int i=0;i<symbol->getArity(); i++) {
    (*sig)[i] = ((CCTerm*)args[i])->getEqClass();
  }
  return sig;
}
```

Retrieving the equivalence class of a node is just a matter of ascending in the tree of the equivalence class until the root and thereby the canonical representative is reached. This is done by the getEqClass method, thereby implementing the find:

```
inline EqClass CCTerm::getEqClass() {
  if(equivClass != this) {
    equivClass = equivClass->getEqClass();
  }
  return equivClass;
}
```

Note that “on the way up“ we are setting the parent of each node we pass to be the root. This technique, known as *path compression*, speeds up the access on subsequent calls of getEqClass on these nodes!

If the node is also a representative of an equivalence class, i.e., it is the root of some equivalence class tree, then the method unionEqClass takes another node being also an equivalence class representative and merges it into the equivalence class of the current node:

```
inline void CCTerm::unionEqClass(EqClass eq) {
  weight += eq->weight;
  eq->equivClass = this;
  predList->splice(predList->end(), *(eq->predList));
  delete eq->predList;
}
```

The first line just adds up the weights of the equivalence classes, the second one makes the other node a child of the current node in the equivalence class tree, and the last two lines just merge the two predecessor lists.

Also the fields predList, weight and auxMember do only have a meaningful value if the node is a representative of an equivalence class. This fact can be queried by the isEqClass method:

```
inline bool CCTerm::isEqClass() const{
  return equivClass == this;
}
```

Representing ground Σ-equations is then an easy task. We chose to make the corresponding class Equation parametric w.r.t. the type of the term, so that we can build equations of CCTerm objects to do the congruence closure and also equations of Term objects to do the decision of an equation.

```
template<class T> class Equation {
public:
  Equation(T* lhs, T* rhs);
  virtual ~Equation();

  T*const getLhs() const;
  T*const getRhs() const;

private:
  T*const lhs;
  T*const rhs;
};
```

Now we can represent sets of equations. This is done by defining a class Theory which basically just contains an array of objects of type Equation<CCTerm>, an ArityTable and a SignatureTable instance.

```
class Theory {
public:
  virtual ~Theory();
  SigTab* copySigTab() const;
  static Theory* parse(istream& input);
  void buildClosure();

private:
  static void swap(EqClass& c1, EqClass& c2);
  void combineAll(Combine& combine,Pending& pending);
  void insertToSignatureTable(CCTerm* t, Combine& combine);
  void traverse(CCTerm* t,Combine& combine);

  ArityTable arityTable;
  int axiomsCount;
  CCEquation** axioms;

  SignatureTable signatureTable;
};
```

The buildClosure method performs the congruence closure algorithm as described in the previous section:

```
void Theory::buildClosure() {
  Combine combine;
  signatureTable = SignatureTable();
  // traverse axioms
  for(int i=0;i<axiomsCount;i++) {
    traverse(axioms[i]->getLhs(),combine);
    traverse(axioms[i]->getRhs(),combine);
  }
  Pending pending;
  combineAll(combine,pending);
  combine.clear();
  while(!pending.empty()) {
    for(Pending::iterator it = pending.begin();
      it != pending.end(); it++) {
      insertToSignatureTable(*it,combine);
    }
    pending.clear();
    combineAll(combine,pending);
    combine.clear();
  }
}
```

Combine is the type of lists of EqClass pairs:

```
typedef list< pair<EqClass,EqClass> > Combine;
```

The method traverse initialises the predList of each node and also does the initialisation of the Σ, C-signature table as it was done in the abstract algorithm in the first for loop by recursively traversing every node of every term of the equations:

```
void Theory::traverse(CCTerm* t,Combine& combine) {
  // initialise signatureTable
  insertToSignatureTable(t,combine);
  // initialise predList
  for(int i=0;i<t->getSymbol()->getArity();i++) {
    (*t)[i]->getEqClass()->getPredList()->push_back(t);
    traverse((*t)[i],combine);
  }
```

The method insertToSignatureTable now does exactly what is done in the first for loop in the abstract algorithm. It first checks whether the node's signature is already in the Σ, C-signature table; if not the signature together with the node's equivalence class is added to the table. Otherwise both the considered node's equivalence class and the equivalence class found in the Σ, C-signature table are put in the combine list as a pair unless they are equal:

```
void Theory::insertToSignatureTable(CCTerm* t, Combine& combine) {
  Signature<EqClass>* sig = t->getSignature();
  SignatureTable::iterator it = signatureTable.find(sig);
  EqClass c = t->getEqClass();
  if(it == signatureTable.end()) {
    signatureTable.insert(make_pair(sig,c));
  } else {
    delete sig;
    if(c != it->second)
      combine.push_back(make_pair(c,it->second));
  }
}
```

Let us go back to buildClosure: combineAll does exactly what the second for loop of the abstract algorithm describes:

```
inline void Theory::combineAll(Combine& combine,Pending& pending) {
  for(Combine::iterator it = combine.begin(); it!=combine.end(); it++) {
    EqClass c1 = (*it).first, c2 = (*it).second;
    swap(c1,c2);
    PredList* c1list = c1->getPredList();
    PredList::iterator c1listEnd = c1list->end();
    if (c1->isEqClass() && c2->isEqClass()) {
      for(PredList::iterator it = c1list->begin(); it!=c1listEnd; it++) {
        Signature<EqClass>* sig = (*it)->getSignature();
        signatureTable.erase(sig);
        delete sig;
        pending.push_back((*it));
      }
      c2->unionEqClass(c1);
    }
  }
}
```

The swap method performes the swap of the nodes depending on the weight of the respective equivalence class:

```
inline void Theory::swap(EqClass& c1, EqClass& c2) {
  if(c1->getWeight() > c2->getWeight()) {
      EqClass& c = c1;
      c1 = c2;
      c2 = c;
  }
}
```

Now we can look at the implementation of the decision algorithm. The buildClosure method computed the Σ, C-signature table. The method copySigTab produces an equivalent Σ, C-signature table as an instance of the class SigTab where we use integers to represent equivalence classes instead:

```
typedef hash_map<Signature<int>*,int> IntSignatureTable;

class SigTab {
public:
  SigTab(const SignatureTable& signatureTable);
  virtual ~SigTab();

  Equation<Term>* parseEquation(const string& equation);
  bool decide(Equation<Term>* equation);

private:
  int findEqClass(Term* t);

  ArityTable arityTable;
  IntSignatureTable signatureTable;
  int nextClass;
};
```

Note that it additionally contains necessary information such as the signature in the form of the arityTable and the integer nextClass representing a fresh equivalence class name. This class also comprises the decision algorithm:

```
bool SigTab::decide(Equation<Term>* equation) {
  return findEqClass(equation->getLhs())
    == findEqClass(equation->getRhs());
}

int SigTab::findEqClass(Term* t) {
  const Symbol* sym = t->getSymbol();
  Signature<int>* sig = new Signature<int>(sym);
  for(int i=0;i<sym->getArity();i++) {
    (*sig)[i] = findEqClass((*t)[i]);
  }
  IntSignatureTable::iterator it = signatureTable.find(sig);
  if(it != signatureTable.end()) {
    delete sig;
    return it->second;
  }
  signatureTable.insert(make_pair(sig, nextClass));
  return nextClass++;
}
```

This is just a straightforward implementation of the algorithm given in figure 2.

5 Complexity Considerations

Now that we have given a concrete implementation of both algorithms we can consider their respective time complexity. At first we will consider the congruence closure algorithm. For this we assume that the input ground Σ-equations are w.r.t. a fixed and finite signature Σ. Hence, there is a constant maximal arity $k \geq 0$ of Σ, i.e., $\Sigma^{(k)} \neq \emptyset$ and $\Sigma^{(l)} = \emptyset$ for all $l > k$. Therefore, each node of the graph of the input set of ground Σ-equations has at most k successors.

Furthermore for a given input of ground Σ-equations E we take $n = \sum_{s \approx t \in E}(|s| + |t|)$ to be the size of the input. One can easily verify that n is exactly the number of nodes of the graph G_E and hence the number of CCTerm instances produced when parsing the input. In the following we analyse the average time complexity, hence we can assume that access operations on the hashtables take constant time!

It is easy to see that the initialisation, i.e. the traversal through all nodes, can be done in $\mathcal{O}(n)$ time as for each node the signature table operation takes constant time as well as each of the at most k additions of the node to a predList. Also for every node there is at most one find operation, i.e., a call to CCTerm::getEqClass, each of which takes constant time as in the beginning every equivalence class contains at most two elements. In sum there are also only at most n additions of pairs of nodes to combine. What follows now is an alternation of calls to combineAll(combine,pending) and for every element *it in pending to insertToSignatureTable(*it ,combine).

Next we bound the number of additions to pending. For this we have to observe that each node can change its equivalence class at most $\log n$ times. The reason for this is that union is done in a weighted manner, i.e., the smaller equivalence class is merged into the bigger one. Hence, if a node's equivalence class changes it is merged into a bigger one thereby at least doubling its size. This can happen only $\log n$ times per node as the maximal size of an equivalence class is obviously n. A node is added to pending only if one of its successors' equivalence class has changed. Each node has at most k successors. Thus, a single node is added to pending at most $k \log n$ times. In sum there will be at most $n\,k \log n \in \mathcal{O}(n \log n)$ additions to pending. As the number of additions to combine is bounded by the initial n times plus the number of additions to pending we have at most $\mathcal{O}(n \log n)$ additions to combine as well.

It is easy to see that there are at most $n - 1 \in \mathcal{O}(n)$ union operations, i.e., calls to the method CCTerm::unionEqClass, as there are initially at most n equivalence classes and each union operation decrements this number. Accessing an equivalence class' predList and also its weight is bounded by a constant times the number of union operations and hence also by $\mathcal{O}(n)$. The number of find operations, i.e., calls to CCTerm::getEqClass, is bounded by the number of of additions to pending plus the initial n times and hence, is bounded by $\mathcal{O}(n \log n)$.

With the chosen representation each union and also each access to an equivalence class' predList and weight takes constant time. Hence, all union operations and accesses to predLists and weights take $\mathcal{O}(n)$ time. Additionally it can also be shown that the $\mathcal{O}(n \log n)$ find operations require $\mathcal{O}(n \log n)$ time [3]. The reason is that the algorithm can be shown to have a time bound of $\mathcal{O}(n \log n)$ on the equivalence class operations for the alternative representation using lookup tables. However as already mentioned this approach can be shown to never outperform the approach using trees chosen here.

Now consider the operations on the signature table. Both looking up a signature and adding a signature take constant time in the average case and are only performed as many times as there are additions to pending plus the additional n times in the initialisation. Hence, these operations are overall bound in time by $\mathcal{O}(n \log n)$ in the average case.

The finishing conversion of the computed Σ, C-signature to an equivalent one using integers to represent equivalence classes is linear in the size of the Σ, C-signature table, which has at most n entries (at most one for each equivalence class) of constant size.

Hence, in sum the congruence closure algorithm runs in $\mathcal{O}(n \log n)$ time!

The analysis for the decision algorithm is a lot easier. Let $s \approx t$ be the input ground Σ-equation and take $m = |s| + |t|$ to be the size of this input. It is easy to see that m is also the number of nodes produced by parsing the input. For each node SigTab::findEqClass is called exactly once by recursion. The operations on the Σ, C-signature table also take only constant time in the average case. Hence, the decision algorithms runs in $\mathcal{O}(m)$ time. That means in particular its runtime is completely independent of the size of the ground Σ-equation that induced the congruence closure!

6 Conclusion

We have briefly recapitulated the theory behind the word problem of ground Σ-equations and argued that computing the congruence closure of the given equations can in fact solve this problem efficiently. It was then shown how terms can be interpreted as graphs to then use this structure to compute the congruence closure.

For this purpose the necessary data structures such as for equivalence classes and for Σ, C-signature were introduced along with a set of operations on then. This was then used to concisely show an abstract algorithm for computing congruence closures of graphs together with an abstract decision algorithm that uses the output of the this algorithm to efficiently decide the word problem.

Afterwards a concrete and efficient implementation of these algorithms using C++ was presented, by showing basic principles and the choice of data structures. Concludingly an argument for the claimed time complexity of both algorithm was given. It has been shown that the congruence closure of a set of ground equations is computed in $\mathcal{O}(n \log n)$ time in average whereas the decision procedure performs in linear time in the size of the equation that is to be checked, independently of the size of the equations inducing the equational theory.

The presented algorithm differs from the one proposed in [3] basically only in the choice on how to merge a pair of equivalence classes; i.e. the condition that decides whether the first equivalence class is merged into the second or vice versa. Instead of using (the minimality of) the size of the respective predecessor lists as the condition as it was done in [3] we used the size of the equivalence class. This enabled us to really establish the upper time complexity bound of $\mathcal{O}(n \log n)$. Also in practice our choice — i.e., weighted union — outperforms the predecessor list approach as one can see in figure 3. It shows the runtime of the congruence closure algorithm using weighted and unweighted union depending on the size of the input equations. The input equations were randomly generated.

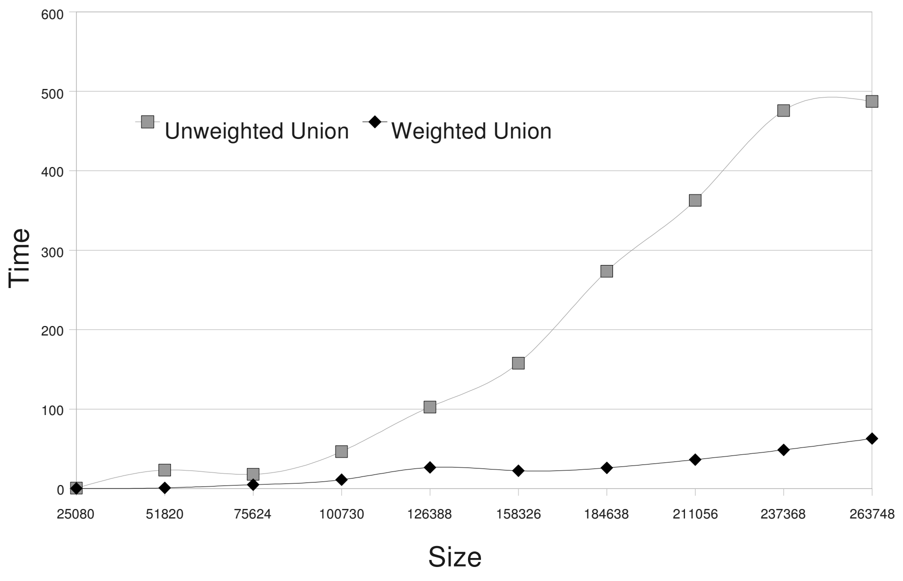

Figure 3: Runtime of the congruence closure algorithm for weighted and unweighted union depending on the input size.

References

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[3] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.