

# A Decision Procedure for Equality Logic

Orly Meir



# A Decision Procedure for Equality Logic

Research Thesis

Submitted in Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science in Computer Science

Orly Meir

Submitted to the Senate of  
the Technion — Israel Institute of Technology  
Haifa

Sivan, 5765

July 2005



The Research Thesis Was Done Under The Supervision of Dr. Ofer Strichman in the Faculty of Computer Science.

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED.



I would like to thank my supervisor Dr. Ofer Strichman for his vast knowledge in the field of verification and for his willingness to share it with me. Thanks for the many hours he invested in teaching me, directing my research and deliberating new ideas with me. Thanks for his mental and moral support throughout this project, and for his encouragement in time of need.

I would also like to thank Sanjit Seshia for the many hours he invested in hooking our procedure to UCLID, and for numerous insightful conversations we had on this and related topics.

# Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Related Work . . . . .	6
<b>2 Preliminaries</b>	<b>9</b>
2.1 Equality Logic and Uninterpreted Functions . . . . .	9
2.2 Reducing Equality Logic to Propositional Logic . . . . .	11
2.3 Equality Graphs . . . . .	14
<b>3 Identifying the necessary transitivity constraints</b>	<b>18</b>
3.1 Monotonicity of NNF formulas . . . . .	18
3.2 A motivating example . . . . .	18
3.3 Main Theorem . . . . .	20
<b>4 The Reduced Transitivity Constraints (RTC) Algorithm</b>	<b>27</b>
4.1 The RTC Algorithm . . . . .	27
4.1.1 Correctness of RTC . . . . .	30
4.1.2 Complexity of RTC and improvements . . . . .	34
4.2 The RTC-EXP Algorithm: An alternative version of RTC . . . . .	35
<b>5 Experimental Results</b>	<b>41</b>
5.1 Optimizations . . . . .	41
5.2 UCLID benchmarks . . . . .	42
5.3 Random graphs . . . . .	43
5.4 Discussion . . . . .	47
<b>6 Conclusions and Future Research</b>	<b>54</b>



<b>A</b>	<b>From Uninterpreted Functions to Equality Logic</b>	<b>55</b>
A.1	Ackermann’s reduction . . . . .	55
A.2	Bryant’s reduction . . . . .	57
<b>B</b>	<b>Example of using Uninterpreted Functions: Translation validation</b>	<b>60</b>
<b>C</b>	<b>Constructing smaller Equality Graphs</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>



# List of Figures

1.1	Pipelined circuit before and after transformation . . . . .	4
2.1	A non-chordal graph and its chordal version . . . . .	14
2.2	Polar Equality Graph . . . . .	15
2.3	A Contradictory Cycle that contains a simple Contradictory Cycle . . . . .	17
3.1	A simple example of our main idea . . . . .	19
3.2	Example for an assignment to Equality Graph's edges . . . . .	22
3.3	Two templates of violating triangles . . . . .	24
4.1	BCCs found by our algorithm . . . . .	30
4.2	Chordal BCCs . . . . .	31
4.3	A chordal Contradictory Cycle . . . . .	32
4.4	An example for redundant constraints . . . . .	35
4.5	Why GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT does not work . . . . .	37
4.6	Why GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT does not work . . . . .	38
4.7	A BCC that causes RTC-EXP to be exponential in the number of vertices. . . . .	40
5.1	Changing solid:dashed ratio: SPARSE vs. RTC (number of constraints) . . . . .	46
5.2	Satisfiable instances: SPARSE vs. RTC . . . . .	48
5.3	Unsatisfiable instances: SPARSE vs. RTC . . . . .	49
5.4	A schematic description of $\mathcal{B} \wedge \mathcal{T}^S$ . . . . .	51
5.5	Unsatisfiable core analysis . . . . .	53
C.1	Equality Graph built according to Bryant's reduction . . . . .	64
C.2	Equality Graph built according to a suggested optimization . . . . .	64

# List of Tables

3.1	$\alpha^R$ vs. $\alpha^S$ . . . . .	19
4.1	The progress of GENERATE-CONSTRAINTS . . . . .	29
4.2	The progress of GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT . . . . .	36
4.3	The continuation of the progress of GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT . . . . .	37
4.4	The progress of GENERATE-CONSTRAINTS-EXP . . . . .	39
5.1	UCLID results, Bryant's reduction . . . . .	44
5.2	UCLID results, Ackermann's reduction . . . . .	44
5.3	RTC vs. the SPARSE method in random formulas with different edges ratio . . . . .	45
5.4	RTC vs. the SPARSE method in random satisfiable formulas with different number of vertices . . . . .	47
5.5	RTC vs. the SPARSE method in random unsatisfiable formulas with different number of edges . . . . .	50
5.6	RTC vs. the SPARSE method in unsatisfiable instances and unsatisfiable core analysis . . . . .	53

# Abstract

We introduce a new decision procedure for deciding Equality Logic, a logic fragment frequently used in verification of infinite-state systems. The procedure improves on Bryant and Velev’s SPARSE method [BV00], in which each equality predicate is encoded with a Boolean variable, and then a set of transitivity constraints are added to compensate for the lost of transitivity of equality. We suggest the Reduced Transitivity Constraints (RTC) algorithm, that unlike the SPARSE method, considers the *polarity* of each equality predicate, i.e. whether it is an equality or disequality when the given equality formula  $\varphi^E$  is in Negation Normal Form (NNF). Given this information, we build the Equality Graph corresponding to  $\varphi^E$  with two types of edges, one for each polarity. We then define the notion of *Contradictory Cycles* to be cycles in that graph that the variables corresponding to their edges cannot be simultaneously satisfied due to transitivity of equality. We prove that it is sufficient to add transitivity constraints that only constrain Contradictory Cycles, which results in only a small subset of the constraints added by the SPARSE method. RTC is a polynomial algorithm which finds these necessary transitivity constraints. The formulas we generate are smaller and define a larger solution set, hence are expected to be easier to solve, as indeed our experiments show. Our new decision procedure is now implemented in the UCLID verification system.

# Chapter 1

## Introduction

In this thesis we suggest a SAT-based decision procedure for solving the satisfiability problem of *Equality Logic*.

Equality Logic is a major decidable theory used in verification of infinite-state systems. This logic can be thought of as Propositional Logic where some or all of the atoms are equalities between variables of some infinite type. For example, the formula  $x = y \wedge (y = z \vee (\neg(x = z) \wedge b))$  is a well-formed Equality Logic formula, where  $x, y, z$  are variables of type **Real** and  $b$  is a Boolean variable. The satisfiability problem of Equality Logic, i.e. whether a formula in this logic is satisfiable or not, is known to be NP-complete (a proof of this claim is presented in Section 2.1).

Equality Logic becomes far more useful when combined with Uninterpreted Functions, as these, on the one hand, enable us to model more sophisticated systems in a natural way, and, on the other hand, can be reduced to Equality Logic. Uninterpreted Functions are widely used in Calculus and other branches of Mathematics, but in the context of reasoning and verification, it is mainly used for simplifying proofs. Under certain conditions, Uninterpreted Functions let us reason about systems while ignoring the semantics of some or all functions and predicates, assuming it is not necessary for the proof. Ignoring the function's semantic means to disregard the axioms that the function can be defined by, and treat it as a function with an arbitrary set of axioms. The only axiom that is left is the one that guarantees that the Uninterpreted Function, like any function, is *consistent*, that is, that given the same inputs, it returns the same outputs. We refer to this property as *Functional Consistency*. For example, if there are two instances of the function  $F$ , say

$F(x_i)$  and  $F(x_j)$ , then Functional Consistency requires that the following relation holds:

$$x_i = x_j \rightarrow F(x_i) = F(x_j) \quad (1.1)$$

There are many cases when the consistency of the function is the only thing that is indeed needed for the proof: the formula to be proven is correct for any function. In these cases Uninterpreted Functions simplify the proof significantly, especially when we try to use automatic theorem provers and decision procedures. In particular, Equality Logic formulas with Uninterpreted Functions can be reduced to Equality Logic, which allows us to generate Equality Logic formulas that their validity implies the validity of the original formula. Two types of this reduction, *Ackermann's Reduction* [Ack54] and *Bryant's Reduction* [BGV99], are described in Appendix A.

One example of using Equality Logic with Uninterpreted Functions is proving equivalence between two models. For example, proving equivalence between two versions of a hardware circuit (a standard procedure in the chip-design industry), such as pipelined vs. non-pipelined processors, or performing *Translation Validation*, a process of proving the semantic equivalence of input and output of a compiler. A detailed example of proving equivalence between circuits is shown in Example 1.1. A detailed example of Translation Validation is shown in Appendix B.

**Example 1.1** (Proving equivalence of circuits). *Pipelining is a technique to improve the performance of a circuit, such as a microprocessor. The computation is split up into phases, called pipeline stages. This allows to speed up the computation by making use of concurrent computation, as done in an assembly line in a factory.*

*The clock frequency of a circuit is limited by the length of the longest path between latches, which is, in the case of a pipelined circuit, simply the length of the longest path between two stages.*

*Figure 1.1A shows a pipelined circuit. The inputs, denoted by  $in$ , are processed in the first stage. We model the combinational gates between the stages by means of Uninterpreted Functions, denoted by  $C, F, G, H, K$ , and  $D$ . The circuit applies function  $F$  to the inputs  $in$ , and stores the result in latch  $L_1$ . This can be formalized as follows:*

$$L_1 = F(in)$$

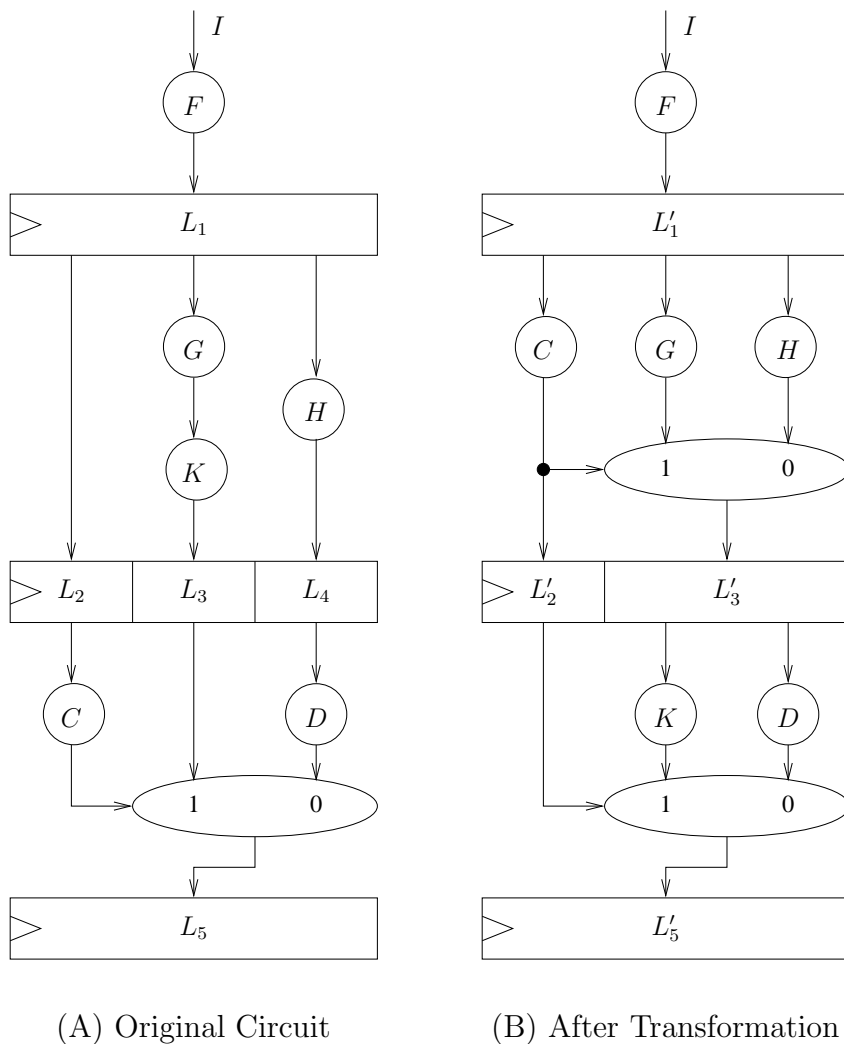


Figure 1.1: Showing correctness of a transformation of a pipelined circuit using Uninterpreted Functions. After the transformation, the circuit has a shorter longest path, and thus, can be operated at a higher clock frequency.



The second stage computes values for  $L_2$ ,  $L_3$ , and  $L_4$ :

$$\begin{aligned} L_2 &= L_1 \\ L_3 &= K(G(L_1)) \\ L_4 &= H(L_1) \end{aligned}$$

The third stage contains a Multiplexer. A Multiplexer is a circuit that selects between two inputs according to the value of a Boolean signal. In this case, this selection signal is computed by a function  $C$ . The output of the multiplexer is stored in latch  $L_5$ .

$$L_5 = C(L_2) ? L_3 : D(L_4)$$

Observe that stage two contains two functions  $G, K$ . The output of  $G$  is used as an input for  $K$ . Suppose this is the longest path within the circuit. We now aim to transform the circuit in order to make it work faster. This can be done in this case by moving the gates represented by  $K$  down into the third stage.

Also observe that only one of the values in  $L_3$  and  $L_4$  is used, as the multiplexer selects one of them depending on  $C$ . We can therefore remove one of the latches by introducing a second multiplexer in stage two. The circuit after these changes is shown in figure 1.1(B). It can be formalized as follows:

$$\begin{aligned} L'_1 &= F(in) \\ L'_2 &= C(L'_1) \\ L'_3 &= C(L'_1) ? G(L'_1) : H(L'_1) \\ L'_5 &= L'_2 ? K(L'_3) : D(L'_3) \end{aligned}$$

The final result of the computation is stored in  $L_5$  in the original circuit, and in  $L'_5$  in the modified circuit. We can show that the transformations preserve the correctness by proving that for all inputs

$$L_5 = L'_5.$$

This can be automated by using a decision procedure for Equalities and Uninterpreted Functions.

The procedure suggested in this thesis improves on Bryant and Velev’s SPARSE method [BV00], in which each equality predicate in the Equality Logic formula is encoded with a Boolean variable, and then a set of *transitivity constraints* is added to compensate for the loss of transitivity of equality. The generated Boolean formula is given to a SAT solver, and its answer implies the satisfiability of the original formula. By considering a different type of a graph we derive a set of constraints that is on the one hand only a small subset of the constraints added by the SPARSE method, and on the other hand is sufficient for generating a formula that is equivalently satisfiable to the original equality formula. The formula we generate is expected to be easier to solve because it defines a much larger solution set. We prove, both theoretically and empirically, that our method is dominant over the SPARSE method, and that this fact is robust with respect to the SAT solver. We suggest a polynomial algorithm, called the Reduced Transitivity Constraints (RTC) algorithm, which finds this necessary subset of transitivity constraints.

The rest of the thesis is organized as follows. In Chapter 2 we show a graph-based reduction called the SPARSE method [BV00], of Equality Logic to Propositional Logic. In Chapter 3 we prove a theorem by which only a subset of the constraints added by this method is necessary. As a result, a Propositional formula with less constraints and a larger solution space can be generated. In Chapter 4 we introduce the Reduced Transitivity Constraints algorithm (RTC), which finds the transitivity constraints that are necessary according to this theorem. In Chapter 5 we present experimental results and a comparison between the SPARSE method and RTC. As expected, when the formula is satisfiable, the Propositional formulas generated by RTC are easier for the SAT solver to solve. When the formula is unsatisfiable the formulas generated by RTC are also easier to solve, although it is not straight-forward to see why (SAT solvers are frequently faster when given further constraints that are consistent with the original formula). We present a possible explanation for this phenomenon and experiments that support it in the end of the chapter. Finally, in Chapter 6 we summarize the work and discuss future research.

## 1.1 Related Work

The importance of Equality Logic led to several suggestions for decision procedures in the last few years, which we will briefly survey here.

The most simple case for solving Equality Logic formulas is solving a conjunction

of equality predicates. This can be done by using equivalence classes, as described in algorithm DECIDE-A-CONJUNCTION-OF-EQUALITIES. The input of this algorithm is a conjunction  $\varphi^E$  of equality predicates and the output is ‘SAT’ if  $\varphi^E$  is satisfiable and ‘UNSAT’ otherwise.

---

Decide-a-conjunction-of-equalities

- 1: Define an equivalence class for each variable. For each equality  $x = y$  unite the equivalence classes of  $x$  and  $y$ .
  - 2: For each disequality  $u \neq v$ : if  $u$  is in the same equivalence class as  $v$  return ‘UNSAT’.
  - 3: Return ‘SAT’.
- 

The more interesting case, however, is solving more efficiently the general case, where the given formula has an arbitrary Boolean structure. The naive approach is to do *Case-splitting*, i.e. convert the formula to DNF-like and solve each DNF clause separately with the procedure described above. This approach is far too costly, as the formula can become exponentially larger than the original one, and solving each of these cases can take a very long time in the worst case. There are far better ways than this naive approach, such as *lazy case-splitting* where rather than pre-generating the DNF, the clauses are generated one at a time, with the hope that satisfiability will be proven before generating and checking all clauses (relying on the observation that satisfying one clause of a DNF is sufficient for proving the satisfiability of the formula). The ability to solve large Equality formulas with arbitrary Boolean structure is a necessity, which motivated through the years the development of procedures which attempt to avoid case splitting. We will concentrate on Binary Decision Diagram (BDD)[Bry86] and SAT-based decision procedures, which are considered to be faster than their predecessors that relied on *lazy case-splitting*.

The first to suggest a BDD based procedure were Goel et al. in [GSZ<sup>+</sup>98]. First, they encode each equality with a new Boolean variable, which results in a formula that we denote by  $\mathcal{B}$  ( $\mathcal{B}$  for *B*oolean). Then, they build a BDD for  $\mathcal{B}$  and search it for a path to the ‘1’ node that does not violate the transitivity of equality. For example, given the formula

$$v_1 = v_2 \wedge v_2 = v_3 \wedge \neg(v_1 = v_3) \tag{1.2}$$

they build the BDD for the formula

$$\mathcal{B} = e_{1,2} \wedge e_{2,3} \wedge \neg e_{1,3} \tag{1.3}$$

where  $e_{1,2}, e_{2,3}, e_{1,3}$  are new Boolean variables<sup>1</sup>. The only path to ‘1’ in the BDD in this case corresponds to the assignment  $(e_{1,2} = T, e_{2,3} = T, e_{1,3} = F)$ , which is of course not consistent with the transitivity of the original formula, and hence discarded. Since there is no other path to ‘1’, the formula is declared unsatisfiable.

The *Range-Allocation* method [PRSS02], offered by Pnueli et al., relies on the Small Model Property, which Equality Logic has. The method suggests to find a small domain for each variable that is sufficient for preserving the satisfiability of the formula. That is, if there is no satisfying assignment from these ranges, then there is no satisfying assignment at all. In the case of the formula above, for example, they find in polynomial time the range  $v_1 \in \{0\}, v_2 \in \{0, 1\}, v_3 \in \{0, 1\}$  to be sufficient. In this specific example no range is needed because the formula is unsatisfiable, but since they invest only polynomial time, they cannot detect that in the general case.

Bryant, German and Velez suggested a related method in [BGV99] called *positive equality*. They classify the terms in the formula to *positive* and *general*. The terms belonging to the first class are those that can be replaced with unique constants without changing the satisfiability of the formula. For example, given the formula  $x \neq y$ ,  $x$  and  $y$  are classified as *positive* since we can check satisfiability of the formula by replacing them with unique constants, for example 0 and 1. Indeed  $0 \neq 1$  is satisfiable as the original formula is. They present a syntactic criterion for this classification that also handles the more complicated case of nested Uninterpreted Functions. A combination of this idea and the Range Allocation method was later suggested by Rodeh and Strichman in [RS01].

The most relevant Decision Procedure for our work is Bryant and Velez’s SPARSE [BV00] method, which we describe in detail in the next chapter.

---

<sup>1</sup> $e_{i,j}$  is the Boolean variable that corresponds to the equality predicate  $v_i = v_j$ . Therefore,  $e_{i,j}$  is equivalent to  $e_{j,i}$ .

# Chapter 2

## Preliminaries

### 2.1 Equality Logic and Uninterpreted Functions

Equality Logic can be thought of as Propositional Logic where some or all the atoms are equalities between variables of some infinite type. More formally:

**Definition 2.1** (Equality Logic). *An Equality Logic formula is defined by the language:*

$$\begin{aligned} \text{formula} & : \text{formula} \vee \text{formula} \mid \neg \text{formula} \mid \text{atom} \\ \text{atom} & : \text{term-variable} = \text{term-variable} \\ & \mid \text{Boolean-variable} \end{aligned}$$

where term-variables are variables defined over some (possibly infinite) domain.

The Equality Logic satisfiability problem is known to be NP-complete, although we are not aware of a proof of this fact in the literature. We prove this fact as follows:

**Theorem 2.1** (Equality Logic satisfiability problem is NP-complete). *The satisfiability problem for Equality Logic is NP-complete.*

*Proof.* We prove that the presented problem is:

1. NP-hard. We introduce a reduction from SAT. Given a Propositional formula  $\mathcal{B}$  with Boolean variables  $b_1 \dots b_n$ , build an Equality Logic formula  $\varphi^{\mathcal{B}}$  by replacing a variable  $b_i, 1 \leq i \leq n$ , with an equality predicate  $x_i = x'_i$ . Now Assume  $\mathcal{B}$  is satisfiable and let

$\alpha$  be a satisfying assignment. We denote by  $\alpha(b_i)$  the value of  $b_i$  under  $\alpha$ . Construct an assignment  $\alpha^E$  as follows (We denote by  $\alpha^E(x_i)$  the value of  $x_i$  under  $\alpha^E$ ):

$$(\alpha^E(x_i), \alpha^E(x'_i)) = \begin{cases} (0, 0) & \alpha(b_i) = \text{TRUE} \\ (0, 1) & \alpha(b_i) = \text{FALSE} \end{cases}$$

Clearly  $\alpha^E$  satisfies  $\varphi^E$ . Assume  $\varphi^E$  is satisfiable and let  $\alpha^E$  be a satisfying assignment. Construct an assignment  $\alpha$  as follows:

$$\alpha(b_i) = \begin{cases} \text{TRUE} & \alpha^E(x_i) = \alpha^E(x'_i) \\ \text{FALSE} & \alpha^E(x_i) \neq \alpha^E(x'_i) \end{cases}$$

Clearly  $\alpha$  satisfies  $\mathcal{B}$ .

2. In NP. We need to show that there exist a non-deterministic Turing machine that can decide this problem. According to [PRSS02], Equality Logic enjoys the Small Model Property, and the smallest solution is bound to be in the range  $1 \dots n$ , where  $n$  is the number of variables in the formula, if the formula is satisfiable. Thus, a non-deterministic Turing machine can scan this range for a solution and check it in polynomial time.

We proved that this problem is NP-hard and in NP, thus it is NP-complete. □

As mentioned in the previous chapter, Equality Logic becomes more useful when combined with Uninterpreted Functions. The formal definition of this combination is as follow:

**Definition 2.2** (Equality Logic with Uninterpreted Functions). *An Equality Logic formula with Uninterpreted Functions is defined by the language:*

$$\begin{aligned} \text{formula} & : \text{formula} \vee \text{formula} \mid \neg \text{formula} \mid \text{atom} \\ \text{atom} & : \text{term} = \text{term} \mid \text{Boolean-variable} \\ \text{term} & : \text{term-variable} \mid \text{function-name}(\text{list of terms}) \end{aligned}$$

## 2.2 Reducing Equality Logic to Propositional Logic

Reduction from Equality Logic to Propositional Logic was first introduced by Goel et al. [GSZ<sup>+</sup>98]. This reduction is useful, because of the availability of highly efficient propositional SAT solvers.

The following framework is used by both [BV00] and our work to reduce this decision problem to the problem of deciding a propositional formula:

1. Let  $E$  denote the set of equality predicates appearing in  $\varphi^E$ . Derive a Boolean formula  $\mathcal{B}$  by replacing each equality predicate  $(v_i = v_j) \in E$  with a new Boolean variable  $e_{i,j}$ . Encode disequality predicates with negations, e.g., encode  $v_i \neq v_j$  with  $\neg e_{i,j}$ .
2. Recover the lost transitivity of equality by conjoining  $\mathcal{B}$  with explicit *transitivity constraints* jointly denoted by  $\mathcal{T}$  ( $\mathcal{T}$  for *Transitivity*).  $\mathcal{T}$  is a formula over  $\mathcal{B}$ 's variables and, possibly, auxiliary variables.

The Boolean formula  $\mathcal{B} \wedge \mathcal{T}$  should be satisfiable if and only if  $\varphi^E$  is satisfiable. Further, we should be able to construct a satisfying assignment to  $\varphi^E$  from an assignment to the  $e_{i,j}$  variables: we would like to construct a satisfying assignment to  $\varphi^E$  by giving to each pair of variables  $v_i, v_j \in \text{vars}(\varphi^E)$  an equal value if  $e_{i,j}$  is set to TRUE, and a different value if  $e_{i,j}$  is set to FALSE.

A straightforward method to build  $\mathcal{T}$  in a way that will satisfy these requirements is the following:

For each set of  $e_{i,j}$  variables  $e_{i_1,i_2}, e_{i_2,i_3}, \dots, e_{i_k,i_1}$  in  $\mathcal{B}$  corresponding to a cyclic comparison between variables in  $\varphi^E$ , add  $k$  *transitivity constraints* of the form

$$\left( \bigwedge_{i=1}^{k-1} \text{antecedent-variable}_i \right) \rightarrow (\text{consequence-variable})$$

where the  $k$  variables in this formula constitute exactly our set. For example, in the constraint

$$e_{i_1,i_2} \wedge e_{i_2,i_3} \dots \wedge e_{i_{k-1},i_k} \rightarrow e_{i_1,i_k} \tag{2.1}$$

we chose  $e_{i_1,i_k}$  to be the consequence variable. By choosing each time a different variable from the set, we get  $k$  constraints.

A transitivity constraint prevents an assignment that contradicts transitivity of equality, that is, that the  $k-1$   $e_{i,j}$  antecedent variables are assigned TRUE, and the remaining single

consequence  $e_{i,j}$  variable is assigned FALSE. The constraint in Equation 2.1, for example, prevents an assignment FALSE to  $e_{i_1,i_k}$  and TRUE to all the other Boolean variables in the constraint. Example 2.1 shows the presented reduction.

**Example 2.1.** *Consider the Equality formula*

$$\varphi^E : v_1 = v_2 \wedge v_2 = v_3 \wedge v_1 \neq v_3$$

*First, we compute  $\mathcal{B}$ :*

$$\mathcal{B} : e_{1,2} \wedge e_{2,3} \wedge \neg e_{1,3}$$

*In this case there is only one cyclic comparison of size 3, and therefore  $\mathcal{T}$  is a conjunction of three transitivity constraints:*

$$\mathcal{T} = \left( \begin{array}{l} (e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}) \wedge \\ (e_{1,2} \wedge e_{1,3} \rightarrow e_{2,3}) \wedge \\ (e_{2,3} \wedge e_{1,3} \rightarrow e_{1,2}) \end{array} \right)$$

*Thus,  $\mathcal{B} \wedge \mathcal{T}$  is satisfiable if and only if  $\varphi^E$  is satisfiable.*

The presented reduction can yield to a Boolean formula with exponential size. In [BV00] three different methods to build  $\mathcal{T}$  are suggested, all of which are better than the one suggested above. In order to explain these methods, we need to define *chord*, *Chordal Graph* and *Non-Polar Equality Graph*.

**Definition 2.3** (Chord). *A chord of a cycle is an edge that connects two vertices that are not adjacent in the cycle.*

**Definition 2.4** (Chordal Graph). *[Ros70] A Chordal Graph is a graph in which every cycle of length greater than three has a chord.*

In Section 5.1 we will describe various techniques for making a graph chordal.

**Definition 2.5** (Non-Polar Equality Graph). *Given an Equality Logic formula  $\varphi^E$ , the Non-Polar Equality Graph corresponding to  $\varphi^E$  is an undirected graph  $(V, E)$  where each node  $v \in V$  corresponds to a variable in  $\varphi^E$ , and each edge  $e \in E$  corresponds to an equality or disequality predicate in  $\varphi^E$ .*



The graph is called *non-polar* to distinguish it from the *Polar Equality Graph* that we will use later, in which there is a distinction between edges that represent equalities and edges that represent disequalities. We will simply say Equality Graph from now on in both cases, where the meaning is clear from the context.

Equality Graphs (whether polar or non-polar), first introduced in [PRSS99], represent an abstraction of equality formulas. An Equality Graph unite all equality formulas with the same set of equality predicates, disregarding the connectivity between them).

The three methods suggested in [BV00] are:

1. The *Dense* method introduces a variable  $e_{i,j}$  for each pair of variables  $(v_i, v_j)$  in  $\varphi^E$ , regardless whether there is a comparison between them in this formula.  $\mathcal{T}$  is then a conjunction of all constraints of the form  $e_{i,j} \wedge e_{j,k} \rightarrow e_{i,k}$  for all distinct values of  $i, j$  and  $k$ . This yields a formula  $\mathcal{T}$  that is cubic in  $n$ , the number of variables in  $\varphi^E$ .
2. The *Direct* method considers the Equality Graph. It adds  $k$  transitivity constraints for every chord-free cycle of size  $k$  in this graph. A proof that this set of constraints is sufficient can be found in [BV00]. Since the number of chord-free cycles in the Equality Graph can be exponential, the formula can grow accordingly.
3. The SPARSE method begins by adding edges to the graph (and correspondingly auxiliary  $e_{i,j}$  variables) in order to make the graph *chordal*. In such a graph only triangles are chord-free, hence it is sufficient to add three transitivity constraints of the form

$$\begin{aligned} (e_{i,j} \wedge e_{j,k} &\rightarrow e_{i,k}) \\ (e_{i,k} \wedge e_{j,k} &\rightarrow e_{i,j}) \\ (e_{i,j} \wedge e_{i,k} &\rightarrow e_{j,k}) \end{aligned}$$

for each triangle  $(v_i, v_j, v_k)$ .

The SPARSE method is both theoretically and empirically the better one of the three, and also our starting point. We will denote the transitivity constraints generated by the SPARSE method with  $\mathcal{T}^S$  (the superscript S is for SPARSE).

**Example 2.2.** Figure 2.1 presents an Equality Graph before and after making it chordal. The added edge  $e_{0,6}$  corresponds to a new auxiliary variable  $e_{0,6}$  that appears in  $\mathcal{T}^S$  but not in  $\mathcal{B}$ . After making the graph chordal, it contains 4 triangles and hence there are 12

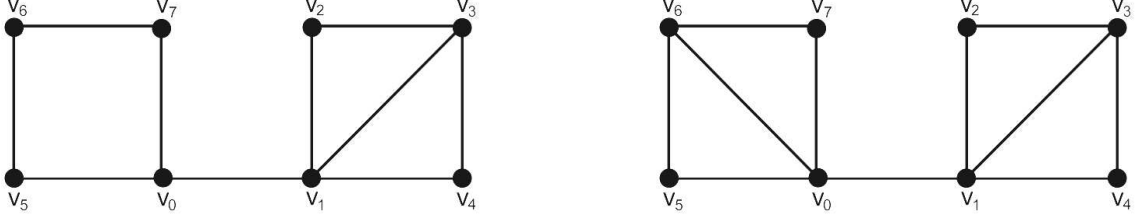


Figure 2.1: A non-chordal Equality Graph corresponding to  $\varphi^E$  (left) and its chordal version (right). The edge  $e_{0,6}$  is added.

constraints in  $\mathcal{T}^S$ . For example, for the triangle  $(v_1, v_2, v_3)$  the constraints are:

$$\begin{aligned} e_{1,2} \wedge e_{2,3} &\rightarrow e_{1,3} \\ e_{1,3} \wedge e_{2,3} &\rightarrow e_{1,2} \\ e_{1,2} \wedge e_{1,3} &\rightarrow e_{2,3} \end{aligned}$$

We will show an algorithm for constructing a Boolean formula  $\mathcal{T}^R$  (the superscript R is for Reduced) which is, similarly to  $\mathcal{T}^S$ , a conjunction of transitivity constraints, but contains only a subset of the constraints in  $\mathcal{T}^S$ . The formula  $\mathcal{T}^R$  is not logically equivalent to  $\mathcal{T}^S$ ; it has a larger solution set. Yet it maintains the property that  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable if and only if  $\varphi^E$  is satisfiable, as we will later prove. This means that not only that  $\mathcal{T}^R$  has a subset of the constraints of  $\mathcal{T}^S$ , it also defines a less constrained search space (i.e. it has more solutions than  $\mathcal{T}^S$ ). Together these two properties are supposed to make the SAT instance easier to solve. Since the complexity of both our algorithm and the SPARSE method are similar, we claim for dominance over the SPARSE method, although practically, due to the unpredictability of SAT, such claims are never 100% true. We will describe this method in detail as of the next section.

## 2.3 Equality Graphs

We assume that our Equality formula  $\varphi^E$  is given in Negation Normal Form (NNF), which means that negations are only applied to atoms, or equality predicates in our case. If a predicate is negated, we say that the polarity of the predicate is negative, and add it to a set that we denote by  $E_{\neq}$ . Otherwise, we say that the polarity of the predicate

is positive, and add it to a set that we denote by  $E_+$ . It is possible, of course, that a predicate appears in both sets. Our decision procedure, as the SPARSE method, relies on graph-theoretic concepts. We will also use Equality Graphs, but redefine it so it refers to polarity information. Specifically, each of the sets  $E_+$  and  $E_-$  corresponds in this graph to a different set of edges. We overload these notations so it refers both to the set of predicates and to the edges that correspond to them in the Equality Graph.

**Definition 2.6** (Polar Equality Graph). *Given an Equality Logic formula  $\varphi^E$ , the Polar Equality Graph corresponding to  $\varphi^E$ , denoted by  $G^E(\varphi^E)$ , is an undirected graph  $(V, E_+, E_-)$  where each node  $v \in V$  corresponds to a variable in  $\varphi^E$ , and each edge in  $E_+$  and  $E_-$  corresponds to an equality or disequality from the respective equality predicates sets  $E_+$  and  $E_-$ . By convention  $E_+$  edges are dashed and  $E_-$  edges are solid.*

As before, every edge in the Equality Graph corresponds to a variable  $e_{i,j} \in \mathcal{B}$ . It follows that when we refer to an assignment of an edge, we actually refer to an assignment to its corresponding Boolean variable. It also follows that the edge  $e_{i,j}$  is equivalent to the edge  $e_{j,i}$ . Notice that an edge can be 'double': both solid and dashed. We will simply use  $G^E$  to denote an Equality Graph if we do not refer to a specific formula.

**Example 2.3.** *In Figure 2.2 we show an Equality Graph  $G^E(\varphi^E)$  corresponding to the non-polar version shown in Figure 2.1, assuming some Equality Formula  $\varphi^E$  for which  $E_+ : \{(v_5 = v_6), (v_6 = v_7), (v_7 = v_0), (v_1 = v_2), (v_2 = v_3), (v_3 = v_4)\}$  and  $E_- : \{(v_0 \neq v_5), (v_0 \neq v_1), (v_1 \neq v_4), (v_1 \neq v_3)\}$ .*

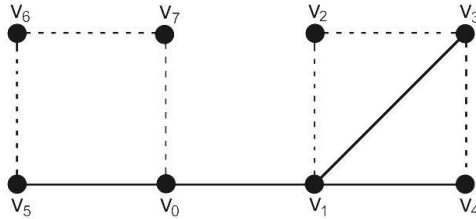


Figure 2.2: The Equality Graph  $G^E(\varphi^E)$  corresponding to the non-polar version of the graph shown in Figure 2.1.

We now define two types of paths in Equality Graphs.

**Definition 2.7** (Equality Path). An Equality Path in an Equality Graph  $G^E$  is a path made of  $E_=$  (dashed) edges. We denote by  $x =^* y$  the fact that  $x$  has an Equality Path to  $y$  in  $G^E$ , where  $x, y \in V$ .

**Definition 2.8** (Disequality Path). A Disequality Path in an Equality Graph  $G^E$  is a path made of  $E_=$  (dashed) edges and a single  $E_{\neq}$  (solid) edge. We denote by  $x \neq^* y$  the fact that  $x$  has a Disequality Path to  $y$  in  $G^E$ , where  $x, y \in V$ .

Similarly, we will use a *Simple Equality Path* and a *Simple Disequality Path* when each vertex along the path is distinct. In Figure 2.2 it holds, for example, that  $v_0 =^* v_6$  due to the simple path  $v_0, v_7, v_6$ . It also holds that  $v_0 \neq^* v_6$  due to the simple path  $v_0, v_5, v_6$ ; and  $v_6 \neq^* v_7$  due to the simple path  $v_6, v_5, v_0, v_7$ .

Intuitively, Equality Path between two variables  $u$  and  $v$  in an Equality Graph  $G$ , implies that there exist a satisfiable equality formula  $\varphi^E$ , such that  $u$  and  $v$  must be equal in order to satisfy  $\varphi^E$ , and  $G^E(\varphi^E) = G$ . A Disequality Path between two variables  $u$  and  $v$  in an Equality Graph  $G$ , implies that there exist a satisfiable equality formula  $\varphi^E$ , such that  $u$  and  $v$  must be assigned different values in order to satisfy  $\varphi^E$ , and  $G^E(\varphi^E) = G$ . For this reason the case in which both  $x =^* y$  and  $x \neq^* y$  hold in  $G^E(\varphi^E)$ , requires special attention. We say that the graph, in this case, contains a *Contradictory Cycle*.

**Definition 2.9** (Contradictory Cycle). A Contradictory Cycle in an Equality Graph is a cycle with exactly one disequality (solid) edge.

Several characteristics of Contradictory Cycles are: 1) For every pair of nodes  $x, y$  in a Contradictory Cycle, it holds that  $x =^* y$  and  $x \neq^* y$ . 2) For every Contradictory Cycle  $C$ , either  $C$  is *simple* or a subset of its edges forms a simple Contradictory Cycle. We will therefore refer only to simple Contradictory Cycles from now on. 3) It is impossible to satisfy simultaneously all the predicates that correspond to edges of a Contradictory Cycle. Further, this is the only type of subgraph with this property [PRSS99].

Figure 2.3 demonstrates characteristic 2: a Contradictory Cycle  $C : (x_1, x_5, x_2, x_3, x_5, x_4, x_1)$  is shown. The cycle  $(x_1, x_5, x_4, x_1)$  is a simple Contradictory Cycle made of a subset of the edges of  $C$ .

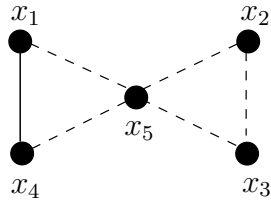


Figure 2.3: Dashed edges are  $E_ =$  edges (equalities) and solid edges are  $E_ \neq$  edges (disequalities). This figure demonstrates that Contradictory Cycles contain a simple Contradictory Cycle: in this case the Contradictory Cycle  $x_1, x_5, x_2, x_3, x_5, x_4, x_1$  contains  $x_1, x_5, x_4, x_1$  which is Contradictory as well.

# Chapter 3

## Identifying the necessary transitivity constraints

### 3.1 Monotonicity of NNF formulas

In the previous chapter we defined the Polar Equality Graph, which is built according to the polarity of the equality predicates in the NNF formula. The reason that we need polarity information, is that it allows us to use the following property of NNF formulas.

**Theorem 3.1** (Monotonicity of NNF). *Let  $\phi$  be an NNF formula and  $\alpha$  be an assignment s.t.  $\alpha \models \phi$ . Let the positive set  $S$  of  $\alpha$  be the positive literals in  $\phi$  assigned TRUE and the negative literals in  $\phi$  assigned FALSE. Every assignment  $\alpha'$  with a positive set  $S'$  s.t.  $S \subseteq S'$  satisfies  $\phi$  as well.*

We are not aware of the first reference to this simple observation, but it was used, for example, in [PRSS02].

We show how this characteristic is being exploited in the next two sections.

### 3.2 A motivating example

We claim that  $\mathcal{T}^S$  contains redundant constraints. In order to make the verification process faster, we want to generate a formula  $\mathcal{T}^R$ , which is also a conjunction of transitivity constraints, such that:

1.  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable if and only if  $\varphi^E$  is satisfiable.

	$\alpha^R$	$\alpha^S$	$\alpha^R$	$\alpha^S$
$e_{0,1}$	TRUE	TRUE	TRUE	FALSE
$e_{1,2}$	TRUE	TRUE	FALSE	FALSE
$e_{0,2}$	FALSE	TRUE	TRUE	TRUE

Table 3.1:

2.  $\mathcal{B} \wedge \mathcal{T}^R$  is easier to solve than  $\mathcal{B} \wedge \mathcal{T}^S$ .

We would like to achieve this goal by generating the formula  $\mathcal{T}^R$  such that it contains less redundant constraints than  $\mathcal{T}^S$ .

The key idea is formulated by Theorem 3.2, which is presented later. This idea can first be demonstrated by a simple example.

**Example 3.1.** *For the Equality Graph below, the SPARSE method generates  $\mathcal{T}^S$  with three transitivity constraints (recall that it generates three constraints for each triangle in the graph, regardless of the edges' polarity). We claim, however, that the single transitivity constraint  $\mathcal{T}^R = (e_{0,2} \wedge e_{1,2} \rightarrow e_{0,1})$  is sufficient.*

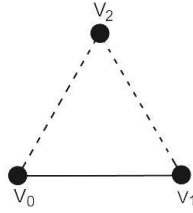


Figure 3.1: A simple example of our main idea

To justify this claim, it is sufficient to show that for every assignment  $\alpha^R$  that satisfies  $\mathcal{B} \wedge \mathcal{T}^R$ , there exists an assignment  $\alpha^S$  that satisfies  $\mathcal{B} \wedge \mathcal{T}^S$ . Since this, in turn, implies that  $\varphi^E$  is satisfiable as well, we get that  $\varphi^E$  is satisfiable if and only if  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable. Note that the ‘only if’ direction is implied by the fact that  $\mathcal{T}^R$  contains a subset of the constraints defined by  $\mathcal{T}^S$ .

We are able to construct such an assignment  $\alpha^S$  because of the monotonicity of NNF (recall that the polarity of the edges in the Equality Graph are according to their polarity in

the NNF representation of  $\varphi^E$ ). There are only two satisfying assignments to  $\mathcal{T}^R$  that do not satisfy  $\mathcal{T}^S$ . Both of these assignments are shown in the  $\alpha^R$  columns in table 3.1. The columns denoted by  $\alpha^S$  show the corresponding assignments  $\alpha^S$ , which clearly satisfies  $\mathcal{T}^S$ . We still need to prove that every formula  $\mathcal{B}$  that corresponds to the above graph, is still satisfied by the assignments in columns  $\alpha^S$  if it was satisfied by the assignments in columns  $\alpha^R$ . For example, for  $\mathcal{B} = (\neg e_{0,1} \vee e_{1,2} \vee e_{0,2})$ ,  $\alpha^R \models \mathcal{B} \wedge \mathcal{T}^R$  and  $\alpha^S \models \mathcal{B} \wedge \mathcal{T}^S$  for all 4 columns. Intuitively, this is guaranteed to be true because  $\alpha^S$  is derived from  $\alpha^R$  by flipping an assignment of a positive (un-negated) predicate from FALSE to TRUE ( $e_{0,2}$  in the  $\alpha^R$  assignment that is presented in the left table) or by flipping an assignment to a negated predicate from TRUE to FALSE ( $e_{0,1}$  in the case of the  $\alpha^R$  assignment that is presented in the right table). A formalization of this argument requires a reference to the monotonicity of NNF (Theorem 3.1): Let  $S$  and  $S'$  denote the positive sets of  $\alpha^R$  and  $\alpha^S$  from the first two columns respectively. In this case  $S = \{e_{0,1}, e_{1,2}\}$  and  $S' = \{e_{0,1}, e_{1,2}, e_{0,2}\}$ . Thus  $S \subset S'$  and hence, according to Theorem 3.1,  $\alpha^R \models \mathcal{B} \rightarrow \alpha^S \models \mathcal{B}$ . The same can be shown for the last two columns.

### 3.3 Main Theorem

We need several definitions in order to state our main theorem.

**Definition 3.1** (A constrained Contradictory Cycle). Let  $C = (e_s, e_1, \dots, e_n)$  be a Contradictory Cycle where  $e_s$  is the solid edge. Let  $\psi$  be a formula over the Boolean variables in  $\mathcal{B}$  that encode  $C$ 's edges.  $C$  is said to be constrained by  $\psi$  if an assignment  $(e_s, e_1, \dots, e_n) \leftarrow (F, T, \dots, T)$  contradicts  $\psi$ .

Recall that we denote by  $\mathcal{T}^S$  the formula that imposes transitivity constraints in the SPARSE method, as defined in [BV00] and described in Section 2.2. Further, recall that the SPARSE method works with chordal graphs, and therefore all constraints are over triangles. Our method also starts by making the graph chordal, and the constraints that we generate are also over triangles, although we will not use this fact in Theorem 3.2, in order to make it more general.

Given an equality formula  $\varphi^E$ , we encode it by  $\mathcal{B}$  and build  $G^E(\varphi^E)$  accordingly. We then conjoin  $\mathcal{B}$  with  $\mathcal{T}^R$ , which is defined as follows:

**Definition 3.2** (Reduced Transitivity Constraints formula  $\mathcal{T}^R$ ). The Reduced Transitivity



Constraints (RTC) formula  $\mathcal{T}^R$  is a conjunction of transitivity constraints that maintains these two requirements:

R1 For every assignment  $\alpha^S$ ,  $\alpha^S \models \mathcal{T}^S \rightarrow \alpha^S \models \mathcal{T}^R$  (The solution set of  $\mathcal{T}^R$  includes all the solutions to  $\mathcal{T}^S$ ).

R2  $\mathcal{T}^R$  constrains all the simple Contradictory Cycles in the Equality Graph  $G^E$ .

R1 implies that  $\mathcal{T}^R$  is less constrained than  $\mathcal{T}^S$ . Consider, for example, a chordal Equality graph in which all edges are solid (disequalities): in such a graph there are no Contradictory Cycles and hence no constraints are required. In this case  $\mathcal{T}^R = \text{TRUE}$ , while  $\mathcal{T}^S$  includes three transitivity constraints for each triangle.

**Theorem 3.2** (Main). *An Equality formula  $\varphi^E$  is satisfiable if and only if  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable.*

*Proof.* ( $\Rightarrow$ ) Due to R1, the proof of this direction is trivial: if  $\varphi^E$  is satisfiable then  $\mathcal{B} \wedge \mathcal{T}^S$  is satisfiable and from R1,  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable.

( $\Leftarrow$ ) We now prove that if  $\mathcal{B} \wedge \mathcal{T}^R$  is satisfiable, then so is  $\mathcal{B} \wedge \mathcal{T}^S$ , which by [BV00] also implies that  $\varphi^E$  is satisfiable. Our proof strategy is:

1. Consider a full assignment  $\alpha^R$  such that  $\alpha^R \models \mathcal{B} \wedge \mathcal{T}^R$ .
2. Construct an assignment  $\alpha^S$  for which  $\alpha^S \models \mathcal{B} \wedge \mathcal{T}^S$  holds.
3. From the correctness of the SPARSE method we conclude that  $\varphi^E$  is satisfiable.

In order to show how to construct  $\alpha^S$ , we need the following definitions:

**Definition 3.3** (Transitivity violation of cycles). *A simple cycle of size  $k$  is said to be violating transitivity under a full assignment  $\alpha^R$  if  $\alpha^R$  assigns exactly  $k - 1$  of its edges TRUE.*

Recall that no cycle can be violating under an assignment that satisfies  $\mathcal{B} \wedge \mathcal{T}^S$ . On the other hand such cycles can exist under  $\alpha^R$ , an assignment that satisfies  $\mathcal{B} \wedge \mathcal{T}^R$ .

Although  $\alpha^S$  and  $\alpha^R$  are assignments of Boolean values to edges, it is still convenient to refer to the vertices that these edges connect, as we do in the following definitions.

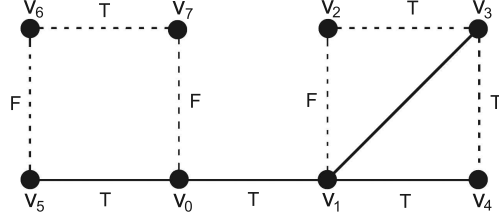


Figure 3.2: The Equality Graph  $G^E(\varphi^E)$  shown in Figure 2.2, with the values that an assignment  $\alpha^R$  assigned to each edge.

**Definition 3.4** (Equal vertices under an assignment). *A pair of vertices  $(v_i, v_j)$  is Equal under an assignment  $\alpha$ , denoted by  $v_i =_{\alpha}^* v_j$ , if and only if there is an Equality Path between them that  $\alpha$  assigns TRUE to all of its edges.*

**Definition 3.5** (Unequal vertices under an assignment). *A pair of vertices  $(v_i, v_j)$  is unequal under an assignment  $\alpha$ , denoted by  $v_i \neq_{\alpha}^* v_j$ , if and only if there is a Disequality Path between them that  $\alpha$  assigns TRUE to all of its dashed (equality) edges and FALSE to its single solid (disequality) edge.*

These definitions can be demonstrated in Figure 3.2: consider an assignment  $\alpha$  which assigns TRUE to  $e_{1,2}$ ,  $e_{2,3}$  and  $e_{3,4}$ , and FALSE to  $e_{1,4}$ . Now  $v_1 =_{\alpha}^* v_4$  because of the Equality Path  $(v_1, v_2, v_3, v_4)$  and  $v_1 \neq_{\alpha}^* v_4$  because of the Disequality Path  $(v_1, v_4)$ .

Given a violating cycle  $C : (e_F = (v_{i_0}, v_{i_1}), e_{T1} = (v_{i_1}, v_{i_2}), \dots, e_{Tk} = (v_{i_k}, v_{i_0}))$  which is respectively assigned by  $\alpha^R$  (FALSE, TRUE,  $\dots$ , TRUE) we make the following key observations:

- O1: The combination of  $v_{i_0} \neq_{\alpha^R}^* v_{i_1}$ ,  $v_{i_1} =_{\alpha^R}^* v_{i_2}, \dots, v_{i_k} =_{\alpha^R}^* v_{i_0}$  implies that  $\alpha^R$  satisfies a Contradictory Cycle, which is impossible by the definition of  $\alpha^R$ . Hence at least one of these relations has to be false.
- O2: If  $e_F$  is solid then  $v_{i_0} \neq_{\alpha^R}^* v_{i_1}$ .
- O3: If  $e_{T1}$  is dashed then  $v_{i_1} =_{\alpha^R}^* v_{i_2}$ , and the same applies for  $e_{T2} \dots e_{Tk}$ : if  $e_{Tj}$  is dashed,  $1 \leq j \leq k$ , then  $v_{i_j} =_{\alpha^R}^* v_{i_{(j+1 \bmod k+1)}}$ <sup>1</sup>.

---

<sup>1</sup>To avoid overly complicated notation we will not add the ‘mod  $k+1$ ’ from now on

We demonstrate these observations with the help of Figure 3.2:

The assignment  $\alpha^R(e_{1,2}, e_{2,3}, e_{3,4}, e_{1,4}) = (\text{TRUE}, \text{TRUE}, \text{TRUE}, \text{FALSE})$  is impossible because it implies that  $\alpha^R$  satisfies a Contradictory Cycle. Therefore by O1 at least one of these assignments is different. Suppose, then, that

$\alpha^R(e_{1,2}, e_{2,3}, e_{3,4}, e_{1,4}) = (\text{FALSE}, \text{TRUE}, \text{TRUE}, \text{TRUE})$ . This is a violating cycle in which  $(e_F, e_{T1}, e_{T2}, e_{T3}) = (e_{1,2}, e_{2,3}, e_{3,4}, e_{1,4})$ . It is *not* the case that  $v_1 \neq_{\alpha^R}^* v_2$  and by O2 indeed  $e_F$  (the edge  $e_{1,2}$ ) is dashed. It is also *not* the case that  $v_1 =_{\alpha^R}^* v_4$  and indeed by O3  $e_{T3}$  (the edge  $e_{1,4}$ ) is solid.

Based on these observations at least one of these two conditions must hold for our violating cycle  $C : (e_F = (v_{i_0}, v_{i_1}), e_{T1} = (v_{i_1}, v_{i_2}), \dots, e_{Tk} = (v_{i_k}, v_{i_0}))$ , as we will later prove in Lemma 3.3:

C1 *not*  $v_{i_0} \neq_{\alpha^R}^* v_{i_1}$ . By observation O2,  $e_F$  must be dashed.

C2 For some  $j$ , *not*  $v_{i_j} =_{\alpha^R}^* v_{i_{j+1}}$ . By observation O3 this implies that  $e_{Tj}$  is solid.

Returning to our example, C2 holds for  $j = 3$  (the edge  $e_{T3}$ , which is  $e_{1,4}$ ) because it is not the case that  $v_1 =_{\alpha^R}^* v_4$ , and indeed  $e_{1,4}$  is solid.

In Figure 3.3 two violating triangles are shown. The condition C2 holds for the left triangle, and condition C1 holds for the right triangle.

The construction of  $\alpha^S$ , described in algorithm RECONSTRUCT- $\alpha^S$ , is based on these two conditions. Initially, for every edge  $e$ ,  $\alpha^S(e) = \alpha^R(e)$ . Then, for every violating cycle, we decide whether to flip a TRUE assignment of a solid edge to FALSE, or a FALSE assignment of a dashed edge to TRUE, until no such cycles are left. Based on the above observations we later prove that this process terminates and indeed constructs an assignment  $\alpha^S$  that satisfies  $\mathcal{B} \wedge \mathcal{T}^S$ . For example, In figure 3.3, in the left triangle we flip the TRUE assignment of the solid edge  $(v_2, v_3)$  to FALSE. In the right triangle we flip the FALSE assignment of the dashed edge  $(v_1, v_3)$  to TRUE.

**Lemma 3.1.** *The reconstruction of  $\alpha^S$  by RECONSTRUCT- $\alpha^S$  terminates.*

*Proof.* We define ‘mis-assigned’ edge to be a solid edge which is assigned TRUE and a dashed edge which is assigned FALSE. In each step, RECONSTRUCT- $\alpha^S$  either stops or flips a TRUE assignments of solid (disequality) edge to FALSE, or a FALSE assignment of dashed (equality) edge to TRUE. This means that unless RECONSTRUCT- $\alpha^S$  terminates, in each

---

**RECONSTRUCT- $\alpha^S$** 

```

1: For every edge  $e$ , Let  $\alpha^S(e) = \alpha^R(e)$ .
2: while TRUE do
3:   Let  $C = (e_F, e_{T_1}, \dots, e_{T_k})$  be a violating cycle under  $\alpha^R$ , for which  $\alpha^R(C) =$ 
   (FALSE, TRUE,  $\dots$ , TRUE). If there is no such cycle then exit.
4:   if condition C1 holds then
5:      $\alpha^S(e_F) = \text{TRUE}$ 
6:   else
7:     if condition C2 holds then
8:        $\alpha^S(e_{T_j}) = \text{FALSE}$   $\triangleright$  the  $j$  from condition C2
9:     else
10:      Abort  $\triangleright$  Lemma 3.3 will prove that we never get here

```

---

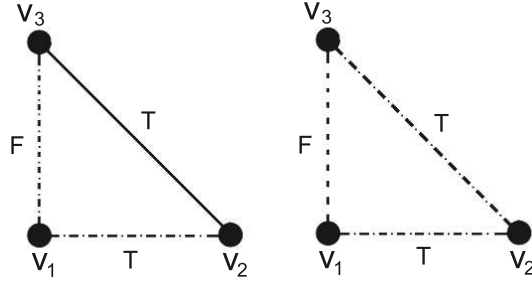


Figure 3.3: Two templates of violating triangles corresponding to conditions C1 and C2. The ‘dashed-dot’ edges  $((v_1, v_2)$  and  $(v_1, v_3)$  in the left triangle and the edges  $(v_1, v_2)$  and  $(v_2, v_3)$  in the right triangle) represent edges that can be either dashed or solid.

of its steps the number of mis-assigned edges is reduced by one. Since there is a finite number of possible mis-assigned edges, the process terminates.  $\square$

**Lemma 3.2.**  $\mathcal{B}$  is satisfied by  $\alpha^S$ .

*Proof.* We only flip assignments of solid (disequality) edges to FALSE, and assignments of dashed (equality) edges to TRUE. Due to monotonicity of NNF formulas (see Theorem 3.1), changing the values in this way preserves satisfiability.  $\square$

**Lemma 3.3.** All violating cycles, at each iteration of RECONSTRUCT- $\alpha^S$ , satisfy one of the two conditions C1 and C2.

*Proof.* By the definition of the two conditions, if a violating cycle does not satisfy any of them, it means that the current assignment satisfies a Contradictory Cycle. We now prove that this is impossible, because it is an *invariant* of RECONSTRUCT- $\alpha^S$  that transitivity of all Contradictory Cycles is not violated.

Assume the reconstruction terminates after  $n$  iterations, and denote by  $\alpha_t$ ,  $1 \leq t \leq n$  the assignment corresponding to the  $t^{th}$  iteration ( $\alpha_1 = \alpha^R$  and  $\alpha_n = \alpha^S$ ). We prove that no Contradictory Cycle can be satisfied, by induction on  $t$ . For  $t = 1$  this holds by definition of  $\alpha^R$ . Now assume this claim is correct for iteration  $t$ ,  $t < n$ . In iteration  $t + 1$  we may:

- Flip the assignment of  $e_F = (v_{i_0}, v_{i_1})$  from FALSE to TRUE. Recall that we perform this flip only if  $e_F$  is dashed. Falsely assume that this flip caused the  $t + 1$ -th assignment to satisfy a Contradictory Cycle  $C$ . This means that  $v_{i_0} \neq_{\alpha_{t+1}}^* v_{i_1}$  and  $v_{i_0} \neq_{\alpha_t}^* v_{i_1}$  due to the path through  $C$ . This contradicts the condition to make this assignment.
- Flip the assignment of  $e_{Tj} = (v_{i_j}, v_{i_{j+1}})$  for some  $1 \leq j \leq k$  from TRUE to FALSE. Recall that we perform this flip only if  $e_{Tj}$  is solid. Falsely assume that this flip caused the  $t + 1$ -th assignment to satisfy a Contradictory Cycle  $C$ . This means that  $e_{Tj}$  is the solid edge of  $C$ . All other edges in  $C$  are dashed and assigned TRUE, and were also assigned TRUE in step  $t$ . This means that  $v_{i_j} =_{\alpha_t}^* v_{i_{j+1}}$  which contradicts the condition to make this assignment.

□

**Lemma 3.4.** *In the end of RECONSTRUCT- $\alpha^S$ , no cycle is violating transitivity under  $\alpha^S$ .*

*Proof.* Due to Lemma 3.3 we know that for every violating cycle one of the two conditions in the reconstruction holds, and hence the truth value of one of its edges is flipped, which immediately makes it stop violating transitivity (recall that by Definition 3.3 a simple cycle with  $k$  vertices is violating transitivity if and only if exactly  $k - 1$  of its edges are assigned TRUE). Hence, there cannot be any violating cycle in the end of this process, which, according to Lemma 3.1, must indeed end. □

The combination of Lemmas 3.2 and 3.4 completes the proof that  $\alpha^S \models \mathcal{B} \wedge \mathcal{T}^S$  and hence proves Theorem 3.2. □

The RECONSTRUCT- $\alpha^S$  algorithm that the proof of Theorem 3.2 relies on, has practical implications. It suggests a way to find a satisfying assignment to  $\varphi^E$ , given  $\alpha^R$  such that  $\alpha^R \models \mathcal{B} \wedge \mathcal{T}^R$ . But RECONSTRUCT- $\alpha^S$  requires searching for violating cycles, which may be exponential. With a small amendment, however, we can make it polynomial. As we will later see in chapter 4, in the process of generating  $\mathcal{T}^R$  we can first make our Equality Graph chordal. We show there, that it is possible to satisfy the requirements from  $\mathcal{T}^R$  with triangular transitivity constraints only (this is also what the SPARSE method does). So by simply adding this as a requirement, that is, that all transitivity constraints constituting  $\mathcal{T}^R$  are triangular, we get a polynomial reconstruction algorithm, since we can now only look for violating triangles, instead of violating cycles. Given this amendment, we can apply RECONSTRUCT- $\alpha^S$  in polynomial time. Finding equality and disequality paths can be done in the following way. Compute two  $n \times n$  matrices (recall that  $n$  is the number of variables in  $\varphi^E$ ), one for Equality Paths and one for Disequality Paths. Both matrices hold, in each  $(i, j)$  entry, a Boolean value. In the Equality Paths matrix, an entry  $(i, j)$  is assigned TRUE if and only if  $v_i =_{\alpha^R}^* v_j$ . In order to build it, initially, mark all entries  $(i, j)$  for which  $(v_i = v_j) \in E_{=}$ . Then, apply a transitive-closure algorithm: for each pair of marked entries  $(i, j)$ ,  $(j, k)$ , mark  $(i, k)$  as well, until convergence. This process can be applied in time cubic in  $n$  [CLR00a]. In the Disequality Paths matrix, an entry  $(i, j)$  is assigned TRUE if and only if  $v_i \neq_{\alpha^R}^* v_j$ . This matrix can be built in a similar way: initially mark all entries  $(i, j)$  for which  $(v_i \neq v_j) \in E_{\neq}$ . Then, mark  $(i, k)$  for each  $k$  such that  $v_j =_{\alpha^R}^* v_k$  and mark  $(j, w)$  for each  $w$  such that  $v_i =_{\alpha^R}^* v_w$ . Then, mark all the pairs  $(w, k)$ , and repeat all this until convergence. This process can also be applied in time cubic in  $n$ .

# Chapter 4

## The Reduced Transitivity Constraints (RTC) Algorithm

### 4.1 The RTC Algorithm

We now introduce an algorithm that generates a formula  $\mathcal{T}^R$ , that satisfies the two requirements R1 and R2 that were introduced in the previous chapter.

The RTC algorithm processes *Biconnected Components* (BCC) in the given Equality Graph.

**Definition 4.1** (Maximal Biconnected Component). *[CLR00b] A Biconnected Component of an undirected graph is a maximal set of edges such that any two edges in the set lie on a common simple cycle.*

From now on we refer to Maximal BCC as BCC.

We can focus on BCCs because we only need to constrain cycles, and in particular Contradictory Cycles. Given an Equality Graph  $G^E$  and a solid edge  $e_s \in G^E$ , we denote by  $G_{=}^E(e_s)$  a graph which contains all the edges of  $E_{=}$  in  $G^E$  and only one solid edge,  $e_s$ . In each step of the loop in line 3 of RTC we handle a different solid edge. For each solid edge  $e_s$ , we first find in line 4, the BCC in  $G_{=}^E(e_s)$  that contains it. We denote this BCC as  $B(e_s)$ . In line 6 of RTC we make  $B(e_s)$  chordal. Since making the graph chordal involves adding edges (and correspondingly new Boolean variables to  $\mathcal{T}^R$ ), we attempt to reduce the number of added edges by relying on existing chords in  $G^E$ . This is the reason we add in line 5  $E_{\neq}$  edges to  $B(e_s)$ . After the graph is chordal we call GENERATE-CONSTRAINTS, which generates and adds to some local cache all the necessary constraints for constraining

all the Contradictory Cycles in this BCC that contain  $e_s$ . When GENERATE-CONSTRAINTS returns, all the constraints that are in the local cache are added to some global cache. The conjunction of the constraints that are in the global cache by the end of the algorithm is what RTC returns as  $\mathcal{T}^R$ .

---

RTC (Equality Graph  $G^E(V, E_-, E_+)$ )

- 1: global-cache =  $\emptyset$
  - 2: Set the source of all edges to NULL
  - 3: **for all**  $e_s \in E_+$  **do**
  - 4:   Find  $B(e_s)$  = maximal BCC in  $G^E$  made of  $e_s$  and  $E_-$  edges;
  - 5:   Add to  $B(e_s)$  all edges from  $E_+$  that connect vertices in  $B(e_s)$ ;
  - 6:   Make  $B(e_s)$  chordal;  $\triangleright$  (The chords can be either solid or dashed)
  - 7:   Clean local cache
  - 8:   GENERATE-CONSTRAINTS ( $B(e_s)$ ,  $e_s$ );
  - 9:   global-cache = global-cache  $\cup$  local-cache;
  - 10:  $\mathcal{T}^R$  = conjunction of all constraints in the global cache;
  - 11: return  $\mathcal{T}^R$ ;
- 

---

GENERATE-CONSTRAINTS (Equality Graph  $G^E(V, E_-, E_+)$ , edge  $e \in G^E$ )

- 1: **for all** triangles  $(e_1, e_2, e) \in G^E$  such that
    - $e_1 \wedge e_2 \rightarrow e$  is not in the local cache
    - $source(e) \neq e_1 \wedge source(e) \neq e_2$
  - do**
  - 2:    $source(e_1) = source(e_2) = e$ ;
  - 3:   Add  $e_1 \wedge e_2 \rightarrow e$  to the local cache;
  - 4:   GENERATE-CONSTRAINTS ( $G^E$ ,  $e_1$ );  $\triangleright$  expand  $e_1$
  - 5:   GENERATE-CONSTRAINTS ( $G^E$ ,  $e_2$ );  $\triangleright$  expand  $e_2$
- 

GENERATE-CONSTRAINTS iterates over all triangles that include the solid edge  $e_s \in E_+$  with which it is called first as a second parameter. It then attempts to implicitly *expand* each such triangle to larger cycles that include  $e_s$ . This expansion is done in the recursive calls of GENERATE-CONSTRAINTS. Given an edge  $e$ , which is part of a cycle, it tries to make the cycle larger by replacing  $e$  with two edges that ‘lean’ on this edge, i.e. two edges  $e_1, e_2$  that together with  $e$  form a triangle. This is why we refer to this operation as expansion. There has to be an indication in which ‘direction’ we can expand the cycle, because otherwise when considering e.g.  $e_1$ , we would replace it with  $e$  and  $e_2$ . For this reason we maintain the *source* of each edge. The *source* of an edge is the edge that it



Iteration	Component	edge to expand	source of edge	Triangle	added constraint
1	a	$e_{0,5}$	-	$(e_{0,5}, e_{5,6}, e_{0,6})$	$e_{0,6} \wedge e_{5,6} \rightarrow e_{0,5}$
2	a	$e_{0,6}$	$e_{0,5}$	$(e_{0,6}, e_{6,7}, e_{0,7})$	$e_{6,7} \wedge e_{0,7} \rightarrow e_{0,6}$
3	b	$e_{1,4}$	-	$(e_{1,4}, e_{3,4}, e_{1,3})$	$e_{1,3} \wedge e_{3,4} \rightarrow e_{1,4}$
4	b	$e_{1,3}$	$e_{1,4}$	$(e_{1,3}, e_{2,3}, e_{1,2})$	$e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$
5	c	$e_{1,3}$	-	$(e_{1,3}, e_{2,3}, e_{1,2})$	$e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$

Table 4.1: The progress of GENERATE-CONSTRAINTS when given the graph of Figure 4.1 (not including steps where the function returns because the triangle contains the source of the expanded edge). In line 5 of the table the constraint  $e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$  is already in the global cache, and hence not added again.

replaces. In the example above when replacing  $e$  with  $e_1, e_2$ ,  $source(e_1) = source(e_2) = e$ . So in the next recursive call, where  $e_1$  is the considered edge, due to the second condition in line 1 of GENERATE-CONSTRAINTS, we *do not* expand it through the triangle  $(e, e_1, e_2)$ .

Each time we replace the given edge  $e$  by two other edges  $e_1, e_2$ , we also add a transitivity constraint  $e_1 \wedge e_2 \rightarrow e$  to the local cache. Informally, one may see this constraint as enforcing the transitivity of the expanded cycle, by using the transitivity enforcement of the smaller cycle. In other words, this constraint guarantees that if the expanded cycle violates transitivity, then so does the smaller one. Repeating this argument all the way down to triangles, gives us an inductive proof that transitivity is enforced for all cycles. A formal proof of correctness of RTC appears in the next subsection.

**Example 4.1.** Figure 4.1 (left) shows the result of the iterative application of line 4 in RTC for each solid edge in the graph shown in Figure 2.2. By the definition of  $B(e_s)$ , after this step each BCC contains exactly one solid edge. Figure 4.1 (right) demonstrates the application of lines 5 and 6 in RTC: in line 5 we add  $e_{1,3}$  to BCC (b), and in line 6 we add  $e_{0,6}$  to BCC (a) (notice that we could choose to add  $e_{5,7}$  as a chord instead), the only additional chords necessary in order to make all BCCs chordal. The progress of GENERATE-CONSTRAINTS for this example is shown in Table 4.1.

The constraints that are generated for this graph are thus:

$$e_{0,6} \wedge e_{5,6} \rightarrow e_{0,5}$$

$$e_{6,7} \wedge e_{0,7} \rightarrow e_{0,6}$$

$$e_{1,3} \wedge e_{3,4} \rightarrow e_{1,4}$$

$$e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$$

#### 4.1.1 Correctness of RTC

**Theorem 4.1** (RTC terminates). *Algorithm RTC terminates.*

*Proof.* In each step in RTC we are handling a solid edge, and all actions taken in lines 4 - 6 clearly terminate. Falsely assume that GENERATE-CONSTRAINTS calls itself an infinite number of times. At each call, GENERATE-CONSTRAINTS either adds a constraint to the local cache if it was not already there, or returns without further recursive calls. There is a finite number of possible constraints (only transitivity constraints over triangles), and therefore at some point all constraints will be in the cache. From that point on, GENERATE-CONSTRAINTS will return without further recursive calls, which contradicts our assumption that it calls itself an infinite number of times.  $\square$

**Lemma 4.1** ( $\mathcal{T}^R$  satisfies requirement R1). *The formula  $\mathcal{T}^R$  returned by RTC respects requirement R1: for every assignment  $\alpha$ ,  $\alpha \models \mathcal{T}^S \rightarrow \alpha \models \mathcal{T}^R$ .*

*Proof.* All the constraints that are added to the cachees are triangular transitivity constraints, and hence in any case we cannot add constraints that would not be otherwise added to  $\mathcal{T}^S$  (to be more precise, this statement is accurate only if we assume that both methods make the given graph chordal in exactly the same way). This means, that the constraints generated by RTC are subset of the constraints generated by the SPARSE method, and therefore the theorem holds.  $\square$

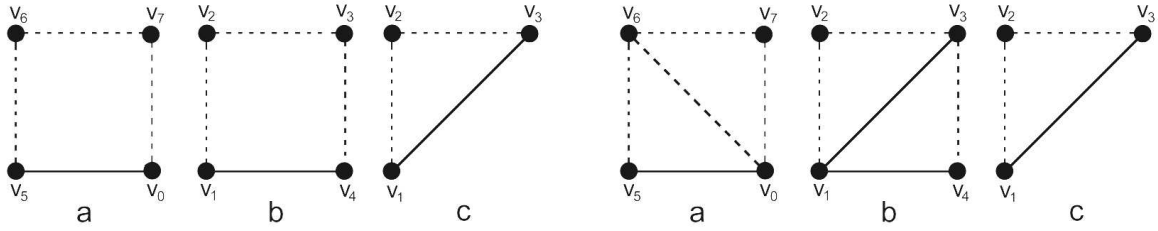


Figure 4.1: The BCCs found in line 4 (left) and after lines 5 and 6 in RTC (right).

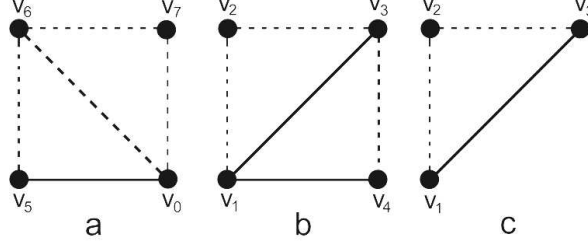


Figure 4.2: The BCCs shown in Figure 4.1, after lines 5 and 6 in RTC

**Lemma 4.2** ( $\mathcal{T}^R$  satisfies requirement R2). *The formula  $\mathcal{T}^R$  returned by RTC respects requirement R2: it constrains all simple Contradictory Cycles in the Equality Graph  $G^E(V, E_-, E_+)$ .*

*Proof.* For each solid edge  $e_s$ , RTC finds  $B(e_s)$ , the maximal BCC that contains  $e_s$  and other dashed edges from  $G^E$ . By the definition of BCC,  $B(e_s)$  contains all simple Contradictory Cycles in which  $e_s$  is the solid edge. In line 8 of RTC, GENERATE-CONSTRAINTS is called with  $B(e_s)$  and  $e_s$  as parameters.

**Proposition 4.1.** *When GENERATE-CONSTRAINTS( $B(e_s)$ ,  $e_s$ ) returns, for every cycle  $C$  containing  $e_s$  the following two properties hold:*

- P1:  $C$  is constrained with respect to  $e_s$  (see Definition 3.1) by the constraints that are in the local cache.*
- P2: for every  $e \in C$  such that  $e \neq e_s$ , a constraint of the form  $e \wedge e_1 \rightarrow e_2$  is in the local cache, such that  $e_1$  and  $e_2$  are either edges on the cycle  $C$  or chords in it.*

*Proof.* By induction over  $n$ , the number of vertices in  $C$ .

**Base case:**  $n = 3$ . When GENERATE-CONSTRAINTS is called from within RTC (and not by a recursive call from GENERATE-CONSTRAINTS itself), each triangle  $C : (e_1, e_2, e_s)$  that contains  $e_s$  is moved over in the loop in line 1, and the constraint  $e_1 \wedge e_2 \rightarrow e_s$  is added to the local cache at line 3, unless it was already in there. This proves both P1 and P2 for triangles.

**Induction step:** Suppose the proposition is correct for cycles with size smaller or equal to  $n$ , and now consider a cycle  $C$  of size  $n + 1$ . As we will later prove in Proposition

4.2, the simple cycle  $C$  can be divided to a cycle  $C_1$  such that  $e_s \in C_1$ , and a triangle  $t : (e_1, e_2, e_3)$ , such that  $e_2, e_3 \in C$  and  $e_1$  is a chord in  $C$  (and also the shared edge between  $C_1$  and  $t$ ) (see Figure 4.3). The induction assumption holds for  $C_1$ . Therefore, since  $e_1$  is an edge on  $C_1$ , a constraint of the form  $e_1 \wedge e_4 \rightarrow e_5$  exists in the local cache, where  $e_4$  and  $e_5$  are edges or chords in  $C_1$ . After adding this constraint,  $source(e_1)$  is set to  $e_5$ , and GENERATE-CONSTRAINTS is called with  $e_1$  as its second parameter. The first parameter is the BCC that contains the cycle  $C$ , and in particular, contains edges  $e_2$  and  $e_3$ . Therefore,  $t$  is being handled in line 1 in GENERATE-CONSTRAINTS. As a result, the constraint  $e_2 \wedge e_3 \rightarrow e_1$  is added to the local cache in line 3 (unless it was already there), since neither  $e_2$  nor  $e_3$  are  $e_1$ 's source. This proves P2 for  $C$ .

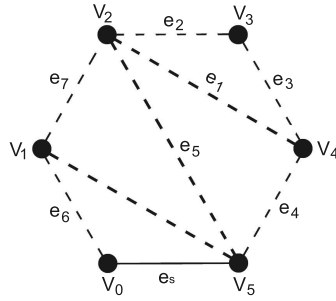


Figure 4.3: A chordal Contradictory Cycle  $C : (e_s, e_6, e_7, e_2, e_3, e_4)$ .  $C$  can be divided to a smaller cycle  $C_1 : (e_s, e_6, e_7, e_1, e_4)$  and a triangle  $t : (e_1, e_2, e_3)$ .

With this additional constraint, we claim that the constraints in the local cache constrain  $C$  with respect to  $e_s$ , which means that P1 holds as well. Falsely assume the contrary, i.e. there exists an assignment which satisfies all the constraints that are in the local cache, that assigns FALSE to  $e_s$  and TRUE to all the other edges in  $C$ . Since the constraint  $e_2 \wedge e_3 \rightarrow e_1$  is in the local cache,  $e_1$  must also be assigned TRUE by this assignment. Now, every edge, except  $e_s$ , in cycle  $C_1$  is assigned TRUE, which contradicts the fact that  $C_1$  is a constrained cycle with respect to  $e_s$ .  $\square$

In order to complete the proof of Proposition 4.1, we need to prove the following Proposition as well:

**Proposition 4.2.** *Let  $C$  be a simple cycle, contained in a chordal BCC, and contains an edge  $e_s$ .  $C$  can be divided to a cycle  $C_1$  such that  $e_s \in C_1$ , and a triangle  $t : (e_1, e_2, e_3)$ , such that  $e_2, e_3 \in C$  and  $e_1$  is a chord in  $C$  (and also the shared edge between  $C_1$  and  $t$ ).*

*Proof.* We use the following definitions and auxiliary propositions:

**Definition 4.2** (Induced Graph). *Given two graphs  $G_1(V_1, E_1)$  and  $G_2 : (V_2, E_2)$ ,  $G_2$  is an Induced Graph of  $G_1$  if  $V_2 \subseteq V_1$  and  $E_2 = \{e : (v_1, v_2) | v_1, v_2 \in V_2 \text{ and } e \in E_1\}$*

**Proposition 4.3.** *Every induced graph of a chordal graph  $G$  is chordal.*

*Proof.* Falsely assume that there exists an induced graph  $G'$  that is not chordal.  $G'$  contains at least one cycle with more than 3 edges, which has no chord. This cycle also exists in  $G$ , in contradiction to the fact that  $G$  is chordal.  $\square$

**Definition 4.3** (Simplicial Vertex). *A vertex  $v \in G$  is simplicial if the neighbors of  $v$  form a clique in  $G$ .*

**Proposition 4.4.** *Every chordal graph  $G$  has a simplicial vertex. If  $G$  is not a complete graph, then it has two simplicial vertices that are not adjacent.*

For proof see, for example, [Shi88].

Every simple cycle  $C$ , together with its chords, is an induced graph of the chordal BCC. Therefore,  $C$  is also a chordal graph, according to the first proposition above. According to the second proposition, If  $G$  is not a complete graph, there are 2 non-adjacent simplicial vertices in it. If  $G$  is a complete graph, then it has at least 3 vertices which are simplicial. In both cases, there is at least one simplicial vertex  $v \in C$ , that is not a vertex on  $e_s$ .  $v$  is part of two edges on  $C$ ,  $e_2 = (v, v_1)$  and  $e_3 = (v, v_2)$ . Since all  $v$ 's neighbors form a clique in the BCC, there is an edge  $e_1 = (v_1, v_2)$ .  $C$  can be divided, therefore, to the cycle  $C_1 = (C \setminus (e_2 \cup e_3)) \cup e_1$  and the triangle  $t : (e_1, e_2, e_3)$ .

This concludes the proof of Proposition 4.2, and hence also the proof of Proposition 4.1.  $\square$

We return to the proof of Lemma 4.2. By Proposition 4.1, when the call to GENERATE-CONSTRAINTS in line 8 returns, all cycles (and in particular, all simple Contradictory Cycles) that contain  $e_s$  are constrained with respect to it by the constraints in the local cache. In line 9 all the constraints in the local cache are added to the global cache. GENERATE-CONSTRAINTS is called for all solid edges, therefore, by the end of the algorithm, for every solid edge, all simple Contradictory Cycles that contain it are constrained with respect to it, by the constraints in the global cache. Since we never remove constraints

from the global cache, we are guaranteed that  $\mathcal{T}^R$ , which conjoins them in line 10 of RTC, constrains all simple Contradictory Cycles. This concludes the proof of Lemma 4.2.  $\square$

**Theorem 4.2** (RTC is sound). *Algorithm RTC generates a formula  $\mathcal{T}^R$  that satisfied requirements R1 and R2 of Theorem 3.2.*

*Proof.* Implied by Lemmas 4.1 and 4.2.  $\square$

### 4.1.2 Complexity of RTC and improvements

Lines 4-5 in RTC can both be done in time linear in the size of the graph (including the process of finding BCCs [CLR00b]). Line 6 can be done in time cubic in  $n$  (full description in Section 5.1). The number of iterations of the main loop in RTC is bounded by the number of solid edges in the graph. GENERATE-CONSTRAINTS, in each iteration of its main loop, either adds a new constraint or moves to the next iteration without further recursive calls. Since the number of possible constraints is bounded by three times the number of triangles in the graph, the number of recursive calls in GENERATE-CONSTRAINTS is bounded accordingly.

*Improvements:* In order to reduce complexity, we use 2 improvements:

1. To avoid the need of making each BCC chordal, for every solid edge, we can start RTC by making the Equality Graph chordal once. This means, by claim 4.3, that each BCC is also chordal, so the rest of the algorithm remains unchanged. In order to avoid creating new Contradictory Cycles in the process of making the graph chordal, we add only solid edges as chords and we avoid going over these new chords in the loop in line 3 of RTC.
2. We only use a global cache, so all the generated constraints are added to one cache only. This reduces the overall complexity, since we never generate the same constraint twice and stop the recursion calls earlier if we encounter a constraint that was generated in a previous BCC. The number of the **total** calls for GENERATE-CONSTRAINTS is therefore bounded by three times the number of triangles in the graph.

The correctness proof of both improvements is very similar to the presented proof of RTC.

## 4.2 The RTC-EXP Algorithm: An alternative version of RTC

As mentioned before, reducing the number of transitivity constraints is our main goal. We have shown that it is enough to constrain only the simple Contradictory Cycles in  $G^E$ . RTC returns  $\mathcal{T}^R$ , which constrains all Contradictory Cycles, whether they are simple or not. Figure 4.4 shows an example of an Equality Graph for which RTC generates redundant constraints. For example, the constraint  $e_{0,2} \wedge e_{0,3} \rightarrow e_{2,3}$  is generated after expanding  $e_{2,3}$  although it is not necessary for constraining any simple Contradictory Cycle.

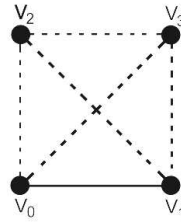


Figure 4.4: An Equality Graph for which RTC generates redundant constraints.

Our first attempt to generate constraints which constrain only simple Contradictory Cycle is shown in GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT algorithm. We observed the need to 'remember' the cycle that we are now expanding, and continue to expand it only if the expansion does not create a non-simple cycle. Function EXPAND-CYCLE-EXP-FIRST-ATTEMPT gets an additional parameter: the cycle that we are now constraining. We then try to expand the cycle to a simple cycle using triangles that contain the edge  $e$ , which is the second parameter of EXPAND-CYCLE-EXP-FIRST-ATTEMPT. Notice that if the cycle that we are trying to expand is not simple, the function returns without adding further constraints.

The problem with this first attempt can be shown in the following example.

**Example 4.2.** Suppose we have the graph shown in figure 4.5. Table 4.2 shows the progress of GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT.

Suppose now that this graph has 3 more edges, as shown in figure 4.6, and the algorithm starts the same way it started in the previous graph, and continues. When we reach

---

GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT (Equality Graph  $G^E(V, E_=: E_{\neq})$ , edge  $e_s \in E_{\neq}$ )

```

1: for all triangles  $C : (e_1, e_2, e_s) \in G^E$  do
2:   if  $e_1 \wedge e_2 \rightarrow e_s$  is not in the cache then
3:     add  $e_1 \wedge e_2 \rightarrow e_s$  to the cache
4:     EXPAND-CYCLE-EXP-FIRST-ATTEMPT( $g, C, e_1$ )
5:     EXPAND-CYCLE-EXP-FIRST-ATTEMPT( $g, C, e_2$ )

```

---



---

Boolean EXPAND-CYCLE-EXP-FIRST-ATTEMPT (Equality Graph  $G^E(V, E_=: E_{\neq})$ , cycle  $C$  that includes  $e_s \in E_{\neq}$ , edge  $e \in C$ )

```

1: for all triangles  $(e_1, e_2, e) \in G^E$  do
2:   if  $(C \setminus e) \cup e_1 \cup e_2$  is a simple cycle and  $e_1 \wedge e_2 \rightarrow e$  is not in the cache then
3:     Add  $e_1 \wedge e_2 \rightarrow e$  to the cache
4:     EXPAND-CYCLE-EXP-FIRST-ATTEMPT( $G^E, (C \setminus e) \cup e_1 \cup e_2, e_1$ )
5:     EXPAND-CYCLE-EXP-FIRST-ATTEMPT( $G^E, (C \setminus e) \cup e_1 \cup e_2, e_2$ )

```

---

Iteration	edge to expand	Triangle	cycle	added constraint	comments
1	$e_{0,1}$	$(e_{0,1}, e_{1,2}, e_{0,2})$	$(e_{0,1}, e_{1,2}, e_{0,2})$	$e_{0,2} \wedge e_{1,2} \rightarrow e_{0,1}$	1
2	$e_{1,2}$	$(e_{1,2}, e_{2,3}, e_{1,3})$	$(e_{0,1}, e_{0,2}, e_{2,3}, e_{1,3})$	$e_{1,3} \wedge e_{2,3} \rightarrow e_{1,2}$	
3	$e_{2,3}$	$(e_{2,3}, e_{3,4}, e_{2,4})$	$(e_{0,1}, e_{0,2}, e_{2,4}, e_{3,4}, e_{1,3})$	$e_{2,4} \wedge e_{3,4} \rightarrow e_{2,3}$	
4	$e_{3,4}$	$(e_{3,4}, e_{4,5}, e_{3,5})$	$(e_{0,1}, e_{0,2}, e_{2,4}, e_{4,5}, e_{3,5}, e_{1,3})$	$e_{4,5} \wedge e_{3,5} \rightarrow e_{3,4}$	
5	$e_{4,5}$	$(e_{4,5}, e_{4,6}, e_{5,6})$	$(e_{0,1}, e_{0,2}, e_{2,4}, e_{4,6}, e_{5,6}, e_{3,5}, e_{1,3})$	$e_{4,6} \wedge e_{5,6} \rightarrow e_{4,5}$	
6	$e_{5,6}$	$(e_{5,6}, e_{3,5}, e_{3,6})$	$(e_{0,1}, e_{0,2}, e_{2,4}, e_{4,6}, e_{3,6}, e_{3,5}, e_{3,5}, e_{1,3})$	-	(*)

Table 4.2: The progress of GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT over the graph shown in figure 4.5. (\*) The constraint  $e_{3,5} \wedge e_{3,6} \rightarrow e_{5,6}$  is not added since the expanded cycle is not simple.





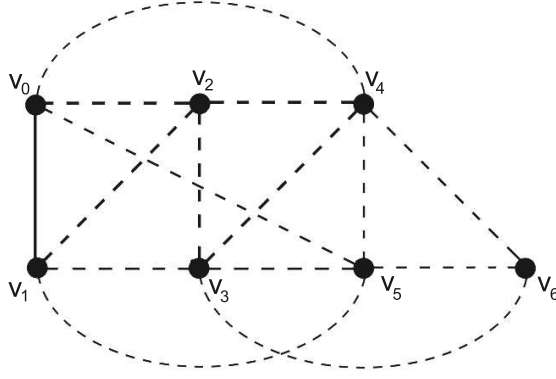


Figure 4.6:

low. GENERATE-CONSTRAINTS-EXP is identical to GENERATE-CONSTRAINTS-EXP-FIRST-ATTEMPT. In EXPAND-CYCLE-EXP, unlike EXPAND-CYCLE-EXP-FIRST-ATTEMPT, the cycle expansion that is implemented by the recursive calls, is done even if the constraint that is found in the current call of the function is already in the cache. This might cause the algorithm to go over the same triangle more than 3 times. In fact, the number of times the algorithm can go over each triangle is bounded by the number of simple Contradictory Cycles, which can be exponential.

---

GENERATE-CONSTRAINTS-EXP (Equality Graph  $G^E(V, E_+, E_-)$ , edge  $e_s \in E_-$ )

- 1: **for all** triangles  $C : (e_1, e_2, e_s) \in G^E$  **do**
  - 2:     **if**  $e_1 \wedge e_2 \rightarrow e_s$  is not in the cache **then**
  - 3:         add  $e_1 \wedge e_2 \rightarrow e_s$  to the cache
  - 4:         EXPAND-CYCLE-EXP( $g, C, e_1$ )
  - 5:         EXPAND-CYCLE-EXP( $g, C, e_2$ )
- 

**Example 4.3.** We now demonstrate how RTC-EXP solves the problem shown in Figure 4.4. For this example, GENERATE-CONSTRAINTS-EXP is called once with the arguments (the whole graph,  $e_{0,1}$ ). Table 4.4 shows step-by-step the progress from that point on. Note how the (redundant) constraint  $e_{0,2} \wedge e_{0,3} \rightarrow e_{2,3}$ , which was added in RTC, is considered in iteration 5 but not added because it is not part of a simple cycle.

As for Example 4.2, the constraint  $e_{3,5} \wedge e_{3,6} \rightarrow e_{5,6}$ , which was not added in EXPAND-CYCLE-EXP-FIRST-ATTEMPT, is added in EXPAND-CYCLE-EXP, since this function does

---

Boolean EXPAND-CYCLE-EXP (Equality Graph  $G^E(V, E_-, E_+)$ , cycle  $C$  that includes  $e_s \in E_+$ , edge  $e \in C$ )

```

1: for all triangles  $(e_1, e_2, e) \in G^E$  do
2:   if  $(C \setminus e) \cup e_1 \cup e_2$  is a simple cycle then
3:     if  $e_1 \wedge e_2 \rightarrow e$  is not in the cache then
4:       Add  $e_1 \wedge e_2 \rightarrow e$  to the cache
                                     ▷ Expand even if the constraint is in the cache
5:     EXPAND-CYCLE-EXP( $G^E$ ,  $(C \setminus e) \cup e_1 \cup e_2, e_1$ )
6:     EXPAND-CYCLE-EXP( $G^E$ ,  $(C \setminus e) \cup e_1 \cup e_2, e_2$ )

```

---

Iteration	edge to expand	Triangle	cycle	added constraint	comments
1	$e_{0,1}$	$(e_{0,1}, e_{1,2}, e_{0,2})$	$(e_{0,1}, e_{1,2}, e_{0,2})$	$e_{0,2} \wedge e_{1,2} \rightarrow e_{0,1}$	1
2	$e_{1,2}$	$(e_{1,2}, e_{0,2}, e_{0,1})$	$(e_{0,1}, e_{0,1}, e_{0,2}, e_{0,2})$	-	2
3	$e_{1,2}$	$(e_{1,2}, e_{2,3}, e_{1,3})$	$(e_{0,1}, e_{0,2}, e_{2,3}, e_{1,3})$	$e_{1,3} \wedge e_{2,3} \rightarrow e_{1,2}$	
4	$e_{2,3}$	$(e_{2,3}, e_{1,3}, e_{1,2})$	$(e_{0,1}, e_{0,2}, e_{1,2}, e_{1,3}, e_{1,3})$	-	2
5	$e_{2,3}$	$(e_{2,3}, e_{0,3}, e_{0,2})$	$(e_{0,1}, e_{0,2}, e_{0,2}, e_{0,3}, e_{1,3})$	-	2
6	$e_{1,3}$	$(e_{1,3}, e_{1,2}, e_{2,3})$	$(e_{0,1}, e_{0,2}, e_{2,3}, e_{2,3}, e_{1,2})$	-	2
7	$e_{1,3}$	$(e_{1,3}, e_{0,1}, e_{0,3})$	$(e_{0,1}, e_{0,2}, e_{2,3}, e_{0,3}, e_{0,1})$	-	2
8	$e_{0,2}$	$(e_{0,2}, e_{0,1}, e_{1,2})$	$(e_{0,1}, e_{0,1}, e_{1,2}, e_{1,2})$	-	2
9	$e_{0,2}$	$(e_{0,2}, e_{0,3}, e_{2,3})$	$(e_{0,1}, e_{0,3}, e_{2,3}, e_{1,2})$	$e_{0,3} \wedge e_{2,3} \rightarrow e_{0,2}$	
10	$e_{0,3}$	$(e_{0,3}, e_{2,3}, e_{0,2})$	$(e_{0,1}, e_{0,2}, e_{2,3}, e_{2,3}, e_{1,2})$	-	2
11	$e_{0,3}$	$(e_{0,3}, e_{1,3}, e_{0,1})$	$(e_{0,1}, e_{0,1}, e_{1,3}, e_{2,3}, e_{1,2})$	-	2
12	$e_{2,3}$	$(e_{2,3}, e_{0,3}, e_{0,2})$	$(e_{0,1}, e_{0,3}, e_{0,3}, e_{2,3}, e_{1,2})$	-	2
13	$e_{2,3}$	$(e_{2,3}, e_{1,3}, e_{1,2})$	$(e_{0,1}, e_{0,3}, e_{1,3}, e_{1,2}, e_{1,2})$	-	2

Table 4.4: The progress of GENERATE-CONSTRAINTS-EXP and EXPAND-CYCLE-EXP when given the graph of Figure 4.4.

*Comments:* 1) The constraint is added in GENERATE-CONSTRAINTS-EXP 2) The considered cycle is not simple: EXPAND-CYCLE-EXP returns without adding a constraint.

not return when getting to the constraint  $e_{4,6} \wedge e_{5,6} \rightarrow e_{4,5}$  in the second time, where it is already in the cache.

The correctness proof for this algorithm is very similar to the correctness proof of RTC. We only need to change the proof of Proposition 4.1, so it refers to simple Contradictory Cycles. Recall that by Theorem 3.2 it is sufficient to constrain these type of cycles. Rather than repeating the proof with small changes, let us give here general arguments that justify the correctness of RTC-EXP.

Proposition 4.1 refers to arbitrary cycles, and is therefore definitely correct for simple cycles. The addition of the condition in line 2 of EXPAND-CYCLE-EXP that requires to consider the triangle  $(e, e_1, e_2)$  only if replacing  $e$  by  $(e_1, e_2)$  maintains  $C$  simple, corresponds to our more modest requirement to constrain only simple cycles. The interesting twist is that blocking the expansion of the cycle can potentially lead to skipping necessary constraints in later steps, unless other changes are made. In particular, note that EXPAND-CYCLE-EXP continues to expand in lines 5 and 6 even if the constraint is already in the cache, in contrast to EXPAND-CYCLE. The reason we need to continue expanding in this case is exactly because of the problem found in EXPAND-CYCLE-EXP-FIRST-ATTEMPT. Unfortunately this necessary remedy causes EXPAND-CYCLE-EXP to be worst-case exponential. Such a case is demonstrated in Figure 4.7. In this graph, there are  $O(2^n)$  ‘paths’ through triangles,  $n$  being the number of squares, but only  $O(n)$  constraints are needed. Since EXPAND-CYCLE-EXP explores all paths regardless of the constraints in the cache, it is exponential.

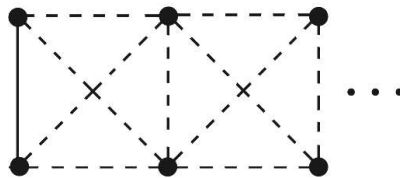


Figure 4.7: A BCC that causes RTC-EXP to be exponential in the number of vertices.

# Chapter 5

## Experimental Results

We first explore two small improvements we used in order to reduce the size of our Equality Graph and the number of generated transitivity constraints.

### 5.1 Optimizations

In order to further reduce the number of transitivity constraints that are generated by our algorithm and to speed up our algorithm, we added the following improvements to our work:

1. *Pre-processing of the input graph.* Our implementation begins with eliminating unnecessary edges and vertices. Since we proved that we only need to constrain simple Contradictory Cycle, we can eliminate from the given Equality Graph vertices and edges that we know that are not part of such cycles. By doing so, we prevent our algorithm from going over those edges and spend time handling them. We eliminate vertices that all of their incident edges are solid, and vertices that their degree is 1.
2. *Heuristics for making the BCC chordal.* In the process of making the BCC chordal in RTC, we add auxiliary Boolean variables. The problem of finding a minimum set of variables to add is shown to be NP-complete in [M81]. In our implementation, as in [BV00] we implement the process of making the graph chordal as a series of elimination steps, starting with graph  $G_0 = G^E$ . On elimination step  $i$ , we create a graph  $G_i$  that is identical to  $G_{i-1}$ , except that some vertex  $m_i$  and its incident edges are removed and new edges are possibly added. In particular, for every pair

of distinct vertices  $j$  and  $k$  such that  $G_{i-1}$  contains the edges  $(m_i, j)$  and  $(m_i, k)$ , we add an edge  $(j, k)$  to  $G_i$ , if it does not already exist. The new chordal graph is the original  $G^E$  plus all the edges that were added in the process. This process is also used in [BV00] for making the graph chordal, in the SPARSE method. To choose which vertex to eliminate on a given step, the SPARSE method uses a known heuristic of choosing the vertex with the minimum degree. In our implementation, we chose to eliminate the vertex that leads to minimum the addition of edges. This heuristic adds less edges than in the SPARSE method.

## 5.2 UCLID benchmarks

Our decision procedure is now integrated in the UCLID [BLS02] verification system. UCLID is a tool for analyzing the correctness of models of hardware and software systems. It can be used to model and verify infinite-state systems with variables of integer, Boolean, function, and array types. The applications of UCLID explored to date include microprocessor design verification, analyzing software for security exploits, verification of a compiler through Translation Validation and verifying distributed algorithms.

UCLID reports to RTC the edges of the Equality Graph corresponding to the verified formula including their polarity, and RTC returns a list of transitivity constraints. The Boolean encoding (the generation of  $\mathcal{B}$ ), the elimination of Uninterpreted Functions, various simplifications and the application of the Positive Equality algorithm [BGV99], are all applied by UCLID as before. The comparison to the SPARSE method of [BV00], which is also implemented in this tool and fed exactly the same formula, is therefore fair.

We used all the relevant UCLID benchmarks that we are aware of (all of which happen to be unsatisfiable). We compared RTC and the SPARSE method using the two different reduction methods of Uninterpreted Functions: Ackermann’s reduction [Ack54] and Bryant’s reduction [BGV99]. This might cause a bias in our results not in our favor: the reduction of Uninterpreted Functions to Equality Logic results in Equality Graphs with specific characteristics which obscure RTC’s advantage. In Appendix A we explained the differences between the two reductions, here we will only say that when Bryant’s reduction is used, all edges corresponding to comparisons between arguments of functions are ‘double’, meaning that they are both solid and dashed. In such a case RTC has no advantage at all, since every cycle is a Contradictory Cycle. This does not mean that when using this reduction

method RTC is useless: recall that we claim for theoretical dominance over the SPARSE method. It only means that the advantage of RTC is going to be visible if there is a large enough portion of the Equality Graph that is not related to the reduction of Uninterpreted Functions, rather to the formula itself.

The SAT-solver we used for both RTC and the SPARSE method was ZCHAFF (2004 edition) [MMZ<sup>+</sup>01]. All benchmarks generated unsatisfiable formulas, thus RTC’s advantage is hard to predict (See Section 5.4 for discussion over this issue). For each benchmark we show the number of generated transitivity constraints, the time it took ZCHAFF to solve the SAT formula, the run time of UCLID, which includes RTC/SPARSE method but not ZCHAFF time, and the total run time. Table 5.1 compares the two algorithms, when UCLID uses Bryant’s reduction with Positive Equality. Indeed, as expected, in this setting the advantage of RTC is hardly visible: the number of constraints is a little smaller comparing to what is generated by the SPARSE method (while the time that takes RTC and the SPARSE method to generate the transitivity constraints is almost identical, with a small advantage to the SPARSE method), and together with the fact that all formulas are unsatisfiable, ZCHAFF runtime is not significantly smaller. We once again emphasize that we consider this as an artifact of the specific benchmarks we found; they are all valid (hence generated unsatisfiable formulas) and almost all equalities in them are associated with the reduction of the Uninterpreted Functions.

As future research we plan to integrate in our implementation the method of Rodeh et al. [RS01] which, while using Bryant’s reduction, not only produces drastically smaller Equality Graphs, but also does not necessarily require a double edge for each comparison of function instances. More explanations about this method can be found in Appendix C.

Table 5.2 compares the two algorithms when Ackermann’s reduction is used. Here the advantage of RTC is seen more clearly, both in the number of constraints and the overall solving times, although the latter is less shown since the formulas are still unsatisfiable. In particular, note the reduction from a total of 222,807 constraints to 67,769 constraints.

## 5.3 Random graphs

In another set of experiments we generated hundreds of random formulas and respective Equality Graphs, while keeping the ratio of vertices to edges similar to what we found in the real benchmarks (about 1 vertex to 4 edges). Each benchmark set was built as follows.

Bench' set	# files	SPARSE method				RTC			
		Const- raints	uclid	zChaff	total	Const- raints	uclid	zChaff	total
TV	9	16719	148.48	1.08	149.56	16083	151.1	0.96	152.06
Cache	4	3669	47.28	40.78	88.06	3667	54.26	38.62	92.88
Dlx1c	3	7143	18.34	2.9	21.24	7143	20.04	2.73	22.77
Elf	3	4074	27.18	2.08	29.26	4074	28.81	1.83	30.64
OOO	6	7059	26.85	46.42	73.27	7059	29.78	45.08	74.86
Pipeline	1	6	0.06	37.29	37.35	6	0.08	36.91	36.99
Total	26	38670	268.19	130.55	398.74	38032	284.07	126.13	410.2

Table 5.1: RTC vs. the SPARSE method using Bryant’s reduction with the Positive Equality optimization. Each benchmark set corresponds to a number of benchmark files in the same family. The column ‘uclid’ refers to the total running time of the decision procedure without the SAT solving time.

Bench' set	# files	SPARSE method				RTC			
		Const- raints	uclid	zChaff	total	Const- raints	uclid	zChaff	total
TV	9	103158	1467.76	5.43	1473.19	9946	1385.61	0.69	1386.3
Cache	4	5970	48.06	42.39	90.45	5398	54.65	44.14	98.79
Dlx1c	3	46473	368.12	11.45	379.57	11445	350.48	8.88	359.36
Elf	5	43374	473.32	28.99	502.31	24033	467.95	28.18	496.13
OOO	6	20205	78.27	29.08	107.35	16068	79.5	24.35	103.85
Pipeline	1	96	0.17	46.57	46.74	24	0.18	46.64	46.82
q2	1	3531	30.32	46.33	76.65	855	32.19	35.57	67.76
Total	29	222807	2466.02	210.24	2676.26	67769	2370.56	188.45	2559.01

Table 5.2: RTC vs. the SPARSE method using Ackermann’s reduction. Each benchmark set corresponds to a number of benchmark files in the same family. The column ‘uclid’ refers to the total running time of the decision procedure without the SAT solving time.



ratio	constraints		zChaff		HaifaSat		siege_v4	
solid:dashed	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC
1:10	373068.8	181707.8	581.1	285.6	549.2	257.4	1321.6	506.4
1:5	373068.8	255366.6	600.0	600.0	600.0	600.0	600.0	600.0
1:2	373068.8	308346.5	600.0	600.0	600.0	600.0	600.0	600.0
1:1	373068.8	257852.6	5.2	0.4	5.9	3.0	1.2	0.1
2:1	373068.8	123623.4	0.1	0.01	0.6	0.22	0.01	0.01
5:1	373068.8	493.9	0.1	0.01	0.6	0.01	0.01	0.01
10:1	373068.8	10.3	0.1	0.01	0.6	0.01	0.01	0.01
average	373068.8	161057.3	255.2	212.3	251.0	208.7	360.4	243.8

Table 5.3: RTC vs. the SPARSE method in random formulas listed by the ratio of solid to dashed edges.

Given  $n$  vertices, we randomly generated 16 different graphs with  $4n$  random edges, and the polarity of each edge was chosen randomly according to a predefined ratio  $p$ . We then generated a random CNF formula  $\mathcal{B}$  with  $16n$  clauses (each clause with up to 4 literals) in which each literal corresponds to one of the edges. Finally, we generated two formulas,  $\mathcal{T}^S$  and  $\mathcal{T}^R$  corresponding to the transitivity constraints generated by the SPARSE and RTC methods respectively, and sent the concatenation of  $\mathcal{B}$  with each of these formulas to three different SAT solvers, HaifaSat [GS05], Siege\_v4 [L.R04] and zChaff 2004.

In the results depicted in Table 5.3 we chose  $n = 200$  (in the UCLID benchmarks  $n$  was typically a little lower than that). Each set of experiments (corresponding to one cell in the table) corresponds to the average results over the 16 graphs, and a different ratio  $p$ , starting from 1 solid to 10 dashed, and ending with 10 solid to 1 dashed. We set the timeout for 600 seconds and added this number in case the solver timed-out. With SIEGE we occasionally let it run until it terminated (with both RTC and SPARSE) without a time limit, just in order to get some information about instances that no solver could solve. All instances were satisfiable, except for the instances with the ratio 1:2 and 1:5 in which we could not solve with any of the solvers, so we cannot determine if they are satisfiable or not. The conclusions from the table are: (1) in all tested ratios RTC generated significantly less constraints than SPARSE. (2) with all three SAT solvers it took longer to solve  $\mathcal{B} \wedge \mathcal{T}^S$  than to solve  $\mathcal{B} \wedge \mathcal{T}^R$ . In some experiments, solvers reached their time limit when solving  $\mathcal{B} \wedge \mathcal{T}^S$ , but solved  $\mathcal{B} \wedge \mathcal{T}^R$  in a few seconds. A visualization of columns 2-3, the number of constraints generated by the two methods, appear in Figure 5.1.

Tables 5.4 and 5.5 present another two sets of experiments. In Table 5.4 all instances are

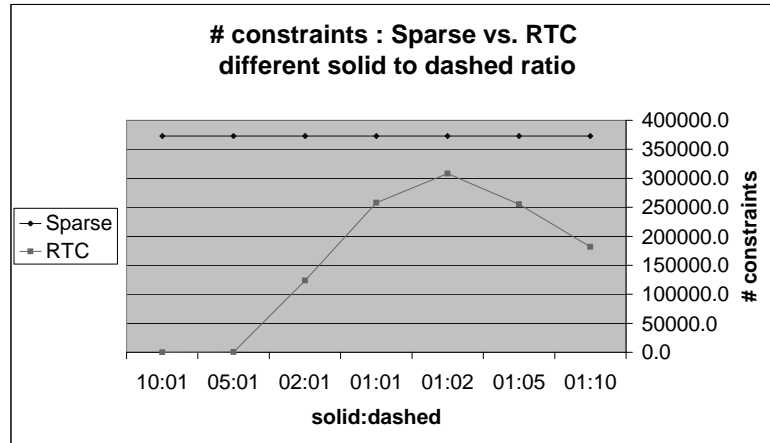


Figure 5.1: Number of constraints as a function of the ratio between solid and dashed edges (corresponding to Table 5.3).

# vertices	constraints		zChaff		HaifaSat		siege_v4	
	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC
200	381622.5	273138.5	0.3	0.1	9.7	2.0	0.4	0.1
300	1231662	881554	4.6	0.5	4.1	2.7	2.1	0.1
400	3005868.7	2183852.7	3.5	0.9	11.0	7.1	9.6	1.1
500	5452234	3774497.7	2.7	1.7	18.8	10.5	1.0	0.2

Table 5.4: RTC vs. the SPARSE method in random graphs with different number of vertices. The table shows for each number of vertices, the average number of constraints and the average time it took each SAT solver to solve the corresponding formula.

satisfiable and in Table 5.5 all instances are unsatisfiable. The tables show the number of constraints generated by the two methods and the average time in seconds that it took each solver to solve the instances. Each line in both tables represent 4 experiments conducted with the same parameters. In Table 5.4, the number of vertices  $n$  ranges from 200 to 500 and the ratio between solid and dashed edges was 1:1. Figure 5.2 shows the results in a more visual way in 2 graphs. The advantage of RTC is seen in both of of these graphs. The average reduction in SAT solving time when using RTC is 63.6%.

Table 5.5 presents results of unsatisfiable instances with  $n = 150$  and 500 to 3000 edges, where each result again averages over 16 different random formulas. Figure 5.3 shows these results in 2 graphs. The number of constraints generated by RTC with these parameters is almost identical to that of the SPARSE method (the large ratio of edges to vertices as well as the 1:1 ratio of solids to dashed edges results in graphs in which many edges are both solid and dashed and belong to some cycle). The advantage of RTC in SAT solving time is rather small (15% time reduction).

As discussed in Section 5.4, the advantage of RTC, shown in Table 5.4, is expected. The experiments above on UCLID benchmarks (which, recall, are all unsatisfiable) and the results from Table 5.5 show that RTC is also better in unsatisfiable instances. A possible reason for this is explained in Section 5.4.

## 5.4 Discussion

It is quite intuitive why the formula  $\mathcal{B} \wedge \mathcal{T}^R$  should be easier to solve than  $\mathcal{B} \wedge \mathcal{T}^S$ , when the original formula  $\varphi^E$  is satisfiable — the solutions space RTC defines is much larger. The case where  $\varphi^E$  is unsatisfiable is more complex to analyze. In the cases where the Boolean

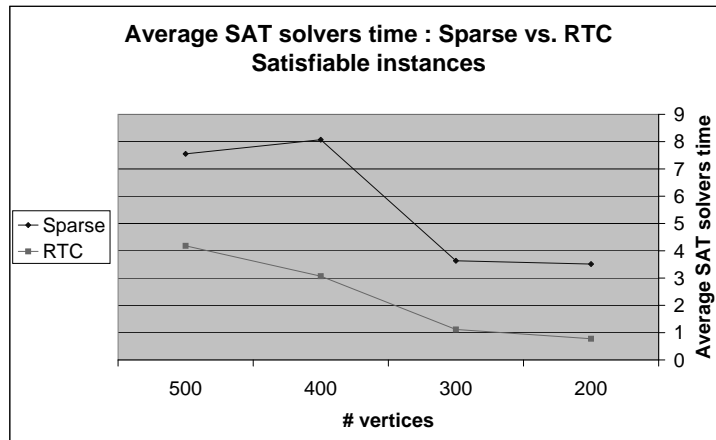
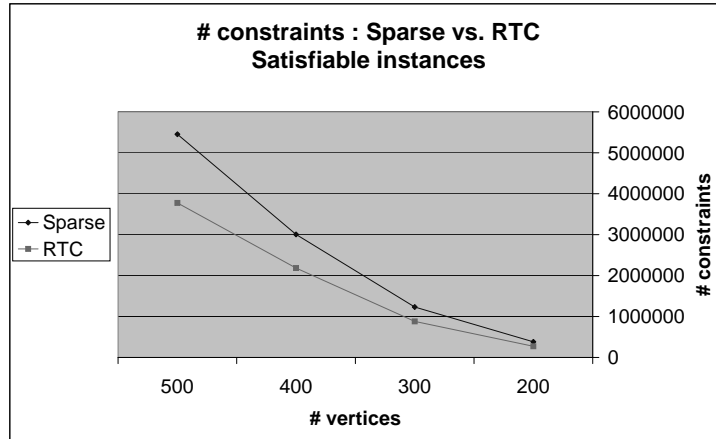


Figure 5.2: Results on satisfiable instances (corresponding to Table 5.4). The number of edges in all the random graphs is 4 times the number of vertices.

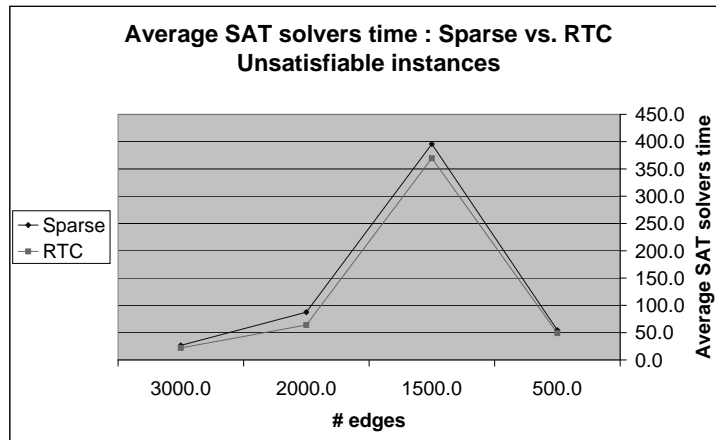
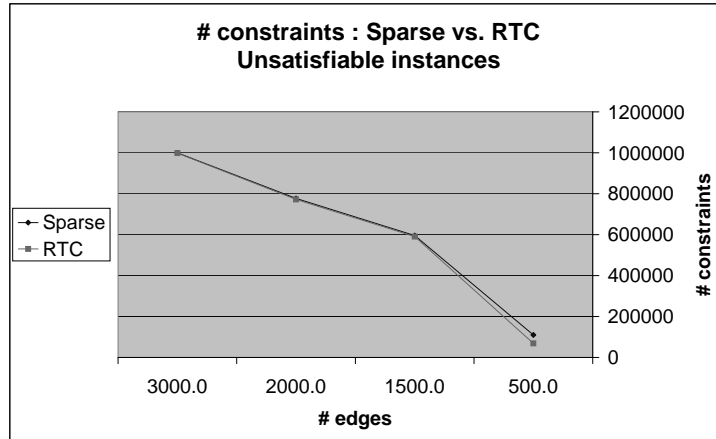


Figure 5.3: Results on unsatisfiable instances (corresponding to Table 5.5). The number of vertices in all the random graphs is 150.

# edges	constraints		zChaff		HaifaSat		siege_v4	
	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC	SPARSE	RTC
500	110139.0	68663.8	50.7	47.6	101.7	89.0	11.2	10.6
1500	595757.3	590674.5	490.4	447.4	565.4	561.3	131.3	100.4
2000	777153	772019.8	134.8	90.7	118.1	92.2	9.6	9.1
3000	999723.8	998443.8	42	34.9	33	27	5.2	4.2

Table 5.5: RTC vs. the SPARSE method in random graphs with 150 vertices and a varying number of edges.

encoding of  $\varphi^E$ ,  $\mathcal{B}$ , is unsatisfiable by itself (without adding any transitivity constraints), we expect that the advantage of RTC will be hardly seen, and  $\mathcal{B} \wedge \mathcal{T}^R$  will be almost as hard to solve as  $\mathcal{B} \wedge \mathcal{T}^S$ . A little advantage of RTC might be seen since  $\mathcal{T}^R$  contains less clauses than  $\mathcal{T}^S$ , and this might cause the SAT solver to waste less time in searching for an unsatisfiability proof in clauses that are not part of such a proof.

In the more interesting cases, where  $\varphi^E$  is unsatisfiable and  $\mathcal{B}$  is satisfiable, it is less clear which of the formulas is easier to solve. In fact, SAT solvers are frequently faster when the input formula contains extra information that further prunes the search space. That might be an advantage of the SPARSE method over RTC. In practice, as can be seen in the experimental results shown in the previous section and in the current section, RTC is better also in unsatisfiable instances. We speculate that this fact originates from the following theorem and conclusions. Let  $T$  represent all transitivity constraints that are in  $\mathcal{T}^S$  but not in  $\mathcal{T}^R$ . Then,

**Theorem 5.1** ( $T$  preserves satisfiability). *If  $\mathcal{B}$  is satisfiable, then  $\mathcal{B} \wedge T$  is satisfiable.*

*Proof.* Since  $\mathcal{B}$  is satisfiable, there is an assignment  $\alpha$  s.t.  $\alpha \models \mathcal{B}$ . Let  $\alpha'$  be an assignment derived from  $\alpha$  such that:

$$\alpha'(e_{i,j}) = \begin{cases} \text{TRUE} & e_{i,j} \in E_-, e_{i,j} \notin E_{\neq} \\ \text{FALSE} & e_{i,j} \in E_{\neq}, e_{i,j} \notin E_- \\ \alpha(e_{i,j}) & \text{Otherwise} \end{cases}$$

We now show that  $\alpha' \models \mathcal{B} \wedge T$ . From monotonicity of NNF formulas, it is clear that  $\alpha' \models \mathcal{B}$ . Falsely assume that  $\alpha'$  does not satisfy  $T$ , i.e. there is a transitivity constraint  $e_{i,j} \wedge e_{j,k} \rightarrow e_{i,k}$  in  $T$  unsatisfied by  $\alpha'$ . This means that  $e_{i,j}$  and  $e_{j,k}$  are assigned TRUE and  $e_{i,k}$  is assigned FALSE in  $\alpha'$ . From  $\alpha'$  definition we know that  $e_{i,j}$  and  $e_{j,k}$  are dashed (or both dashed and solid) and  $e_{i,k}$  is solid (or both solid and dashed), which means that

the triangle  $e_{i,j}, e_{j,k}, e_{i,k}$  is a Contradictory Cycle and hence the constraint  $e_{i,j} \wedge e_{j,k} \rightarrow e_{i,k}$  must be in  $\mathcal{T}^R$ . This contradicts  $T$ 's definition.  $\square$

When an instance is unsatisfiable, most SAT solvers generate a resolution-based proof of unsatisfiability using some or all of the formula's clauses. The drawing below shows a schematic description of the clauses in the CNF formula generated by the SPARSE method.

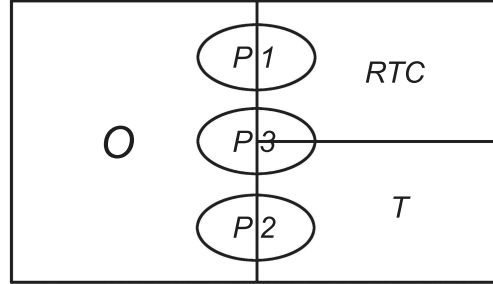


Figure 5.4: A schematic description of the clauses in the CNF formula generated by the SPARSE method.  $P1, P2, P3$  are types of possible unsatisfiability proofs.

The set  $O$  is the set of Original clauses,  $RTC$  is the set of clauses that are generated by both  $RTC$  and the SPARSE method, and  $T$  is the set of clauses defined above. In other words,  $O = \text{cnf}(\mathcal{B})$ ,  $RTC = \text{cnf}(\mathcal{T}^R)$ ,  $T = \text{cnf}(\mathcal{T}^S \wedge \neg \mathcal{T}^R)$  where  $\text{cnf}(\varphi)$  is the CNF representation of a Boolean formula  $\varphi$ .  $P1, P2, P3$  denote different subsets of all the clauses. Assuming  $\mathcal{B}$  is satisfiable, by Theorem 5.1 we know that any proof of unsatisfiability of the formula  $\mathcal{B} \wedge \mathcal{T}^S$  must include  $\mathcal{T}^R$  clauses (note that this also implies that there is no proof based solely on  $O$  clauses). Therefore a proof based on a subset of clauses such as  $P2$  is impossible. We also know from Theorem 3.2, that there is an unsatisfiability proof that includes only  $O$  and  $RTC$  clauses. This means that a proof based on a subset of clauses such as  $P1$  must exist. So a proof can either be of the  $P1$  or  $P3$  type. We hypothesize that a SAT solver typically finds unsatisfiability proofs that do not include many of the  $T$  clauses in the proof (recall that these clauses are not necessary for the proof: we know that there exists a proof without  $T$  clauses at all). Therefore while  $T$  clauses can cause the SAT solver to waste time in solving conflicts that involve them, it is not clear whether they can compensate on this fact by participating in a shorter resolution proof. To check this hypothesis, we ran the following experiment.

We randomly generated 10 graphs with 150 vertices, 1500 edges and a ratio of 1 dashed

edge to 5 solid edges. With these parameters the formula is on the one hand unsatisfiable and on the other hand causes a visible difference in the number of generated constraints and the SAT solving time. Table 5.6 summarizes our results with these graphs. The average number of constraints generated by the SPARSE method is 891,284 and by RTC it is 459,811.1. The average difference in SAT solving time between the two algorithms is 10.913 seconds in favor of RTC. In this table, we show the partition of the whole clauses in the formula to original clauses ( $O$ ),  $T$  clauses and RTC clauses. In addition, we analyzed the unsatisfiable cores generated by ZCHAFF when given the SPARSE method’s formulas. The partition of these cores is shown in the ‘Unsatisfiable core’ columns. We found that all cores include  $T$  clauses, but the percentage of the  $T$  clauses in the cores is smaller than their respective percentage from the whole set of clauses in the original formula.

More formally: let  $O^c$ ,  $RTC^c$  and  $T^c$  represent the set of original, RTC, and  $T$  clauses from the unsatisfiable core, respectively. We can clearly see from the experiments that the following ratio holds:

$$\frac{|T^c|}{|O^c| + |T^c| + |RTC^c|} \ll \frac{|T|}{|O| + |T| + |RTC|}$$

This fact can also be seen in the graph shown in Figure 5.5. The black line shows the average percentage of  $O$ , RTC and  $T$  clauses from the whole set of clauses. The gray line shows the average percentage of  $O^c$ ,  $T^c$  and  $RTC^c$  clauses from the unsatisfiable core. This graph strongly supports our hypothesis: indeed the  $T$  clauses’ participation in the proof is significantly smaller than their percentage in the original formula. Thus, they hardly ‘help’ the SAT solver in finding the unsatisfiability proof while, as the differences in solving times in favor of RTC show, they impose an overhead.



cnf #	zChaff times		CNF				Unsatisfiable Core			
	SPARSE	RTC	Total	Original clauses	$T$	RTC	Total	Original clauses	$T$	RTC
1	37.5	24.84	825818	50000	355200	420618	9175	1948	2182	5045
2	18.35	6.01	979265	50000	492469	436796	3997	1051	1093	1853
3	18.34	8.65	843860	50000	380500	413360	6830	1727	1646	3457
4	14.46	4.29	1183736	50000	571390	562346	4472	1195	1165	2112
5	31.78	18.65	1048711	50000	388899	609812	9055	1908	1732	5415
6	8.47	2.44	901253	50000	471970	379283	2466	753	556	1157
7	11.53	5.8	1043093	50000	529246	463847	3207	1020	643	1544
8	48.11	20.82	820661	50000	349883	420778	2094	801	272	1021
9	12.54	4.76	767009	50000	305619	411390	2376	882	365	1129
10	68.22	38.91	999434	50000	469553	479881	8655	1797	2347	4511
average	22.93	12.017	941284	50000	431472.9	459811.1	5232.7	1308.2	1200.1	2724.4

Table 5.6: RTC vs. SPARSE method in unsatisfiable instances. These results are over random graphs with 150 vertices, 1500 edges and ratio of 1 dashed edge to 5 solid edges. The table shows the number of constraints generated by both algorithms and the SAT solving times of the generated formulas. The 3 last columns are the unsatisfiable core analysis of the formula generated by the SPARSE method.

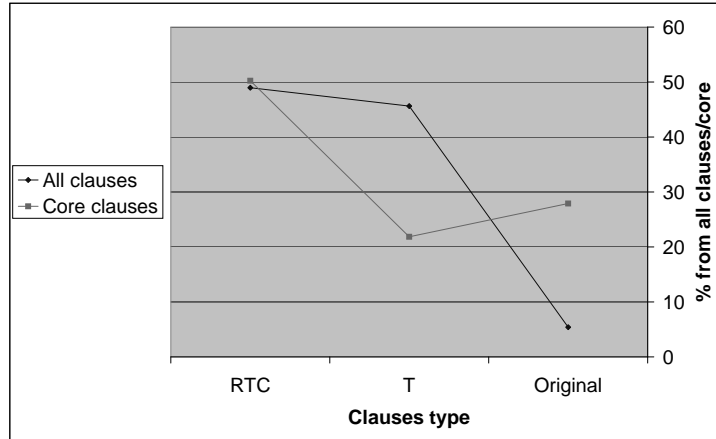


Figure 5.5: The average percentage of original clauses,  $T$ 's clauses and RTC clauses from  $\mathcal{B} \wedge \mathcal{T}^S$  (the black line) and from the unsatisfiable cores (the gray line).

# Chapter 6

## Conclusions and Future Research

We presented a new decision procedure for Equality Logic, which builds upon and improves previous work by Bryant and Velev in [BV00, BGV01]. The new procedure generates a set of transitivity constraints that is, at least in theory, easier to solve. The experiments we conducted show that in most cases it is better in practice as well, and in any case does not make it worse, at least not in more than a few seconds. RTC does not make full use of Theorem 3.2, which states that it is sufficient to constrain simple Contradictory Cycles: it is not able to distinguish between simple and non simple cycles, and hence constrains all cycles, which leads to redundancy. We also presented a second version of the algorithm, RTC-EXP, which indeed constrains only simple Contradictory Cycles, but with a worst-case exponential price. It is left for future research to find a polynomial algorithm that does the same, or prove that this problem is inherently hard.

# Appendix A

## From Uninterpreted Functions to Equality Logic

One way to be able to verify formulas with Uninterpreted Functions, is to reduce them to Equality Logic. We will see two possible reductions, *Ackermann's Reduction* and *Bryant's Reduction*, both of which enforce Functional Consistency. The former is somewhat more intuitive to understand, but also imposes certain restrictions on the decision procedures that can be used to solve it, unlike the latter. In order to simplify the explanations we will only show the version of the reductions for functions with one argument.

### A.1 Ackermann's reduction

Ackermann's reduction captures directly the Functional Consistency condition stated in Equation 1.1.

**Definition A.1** (Ackermann's reduction (simple case)). Input: A formula  $\varphi^{UF}$  with Uninterpreted Functions.

Output: An Equality formula  $\varphi^E$ , which is valid only if  $\varphi^{UF}$  is valid under all interpretations that respect Functional Consistency.

Given a formula  $\varphi^{UF}$  with  $m$  instances of an Uninterpreted Function  $F$ :

$$F(x_1), F(x_2), \dots, F(x_m)$$

where each of the arguments  $x_j$ ,  $1 \leq j \leq m$  is syntactically distinct, and can be either a variable or a function instance, reduce it to an Equality Logic formula  $\varphi^E$  as follows:

1. Derive a subformula  $flat^E$  from  $\varphi^{UF}$  by replacing the  $i^{th}$  function instance,  $1 \leq i \leq m$ , with a new variable  $f_i$ . We refer to  $flat^E$  as the flattened version of  $\varphi^{UF}$ .
2. Construct a subformula  $consistent^E$  by conjoining Functional Consistency constraints as follows: for each pair of function instances that were mapped in the previous step to  $f_i, f_j$ ,  $1 \leq i < j \leq m$  add a constraint

$$(x_i = x_j) \rightarrow f_i = f_j$$

$consistent^E$  conjoines Functional Consistency constraints.

3. Let  $\varphi^E : consistent^E \rightarrow flat^E$ .

To summarize our new notations: given a formula with Uninterpreted Functions  $\varphi^{UF}$  that we wish to validate, Ackermann's reduction reduces it to an Equality Logic formula  $\varphi^E$  of the form:

$$\varphi^E : consistent^E \rightarrow flat^E \tag{A.1}$$

where  $consistent^E$  is a conjunction of Functional Consistency constraints, and  $flat^E$  is a flattening of  $\varphi^{UF}$ , i.e. each function instance  $F_i(x_i)$  in  $\varphi^{UF}$  is replaced with a corresponding new variable  $f_i$ .

**Example A.1.** Consider now the following formula which we wish to validate:

$$x_1 = x_2 \rightarrow F(F(G(x_1))) = F(F(G(x_2)))$$

First, we assign an index to each of the function instance. With Ackermann reduction the index assigned to each function instance is immaterial, but we will nevertheless assign it from the inside out, in order to be consistent with Bryant's reduction that we will consider next (Bryant's reduction indeed requires this order).

$$x_1 = x_2 \rightarrow F_2(F_1(G_1(x_1))) = F_4(F_3(G_2(x_2)))$$

And we now compute:

$$flat^E : \quad x_1 = x_2 \rightarrow f_2 = f_4$$

$$consistent^E : \quad \begin{array}{llll} x_1 = x_2 & \rightarrow & g_1 = g_2 & \wedge \\ g_1 = f_1 & \rightarrow & f_1 = f_2 & \wedge \\ g_1 = g_2 & \rightarrow & f_1 = f_3 & \wedge \\ g_1 = f_3 & \rightarrow & f_1 = f_4 & \wedge \\ f_1 = g_2 & \rightarrow & f_2 = f_3 & \wedge \\ f_1 = f_3 & \rightarrow & f_2 = f_4 & \wedge \\ g_2 = f_3 & \rightarrow & f_3 = f_4 & \end{array}$$

and then, again,

$$\varphi^E : consistent^E \rightarrow flat^E$$

## A.2 Bryant's reduction

*Bryant's reduction* has the same goal as Ackermann's reduction: to reduce formulas with Uninterpreted Functions to Equality Logic.

**Definition A.2** (Bryant's reduction (simple case)). *Given a formula  $\varphi^{UF}$  with  $m$  instances of an Uninterpreted Function  $F$ :*

$$F(x_1), F(x_2), \dots, F(x_m)$$

where each of the arguments  $x_j$ ,  $1 \leq j \leq m$  is syntactically distinct from the other arguments and can be either a variable or a function instance, derive an Equality Logic formula  $\varphi^E$  as follows:

1. Assign indices to the Uninterpreted Function instances  $F_1(), \dots, F_m()$  such that if  $F_i()$  is an argument of  $F_j()$  then  $i < j$ .
2. Derive a subformula  $\varphi_\star^E$  from  $\varphi^{UF}$  by replacing the  $i^{th}$  function instance,  $1 \leq i \leq m$ , with a variable  $F_i^\star$ .
3. Construct a subformula  $consistent^E$  by conjoining, for  $1 \leq i \leq m$ , the definition of  $F_i^\star$ :

$$F_i^\star = \left( \begin{array}{lll} \text{case} & x_1 = x_i & : f_1 \\ & \vdots & \\ & x_{i-1} = x_i & : f_{i-1} \\ & \text{TRUE} & : f_i \\ \text{esac} & & \end{array} \right)$$

where  $f_i$  is a new variable.

4. To check for validity of  $\varphi^{UF}$  let  $\varphi^E = \text{consistent}^E \rightarrow \varphi_\star^E$ . To check for satisfiability of  $\varphi^{UF}$  let  $\varphi^E = \text{consistent}^E \wedge \varphi_\star^E$ .

Finally, let us reconsider the formula of Example A.1:

**Example A.2.**

$$x_1 = x_2 \rightarrow F(F(G(x_1))) = F(F(G(x_2)))$$

and once again when the function instances of each of the function symbol  $F$  and  $G$  are numbered inside out:

$$x_1 = x_2 \rightarrow F_2(F_1(G_1(x_1))) = F_4(F_3(G_2(x_2)))$$

Recall that in Bryant's reduction this order is required. Applying Bryant's reduction we get:

$$\varphi_\star^E : (x_1 = x_2 \rightarrow F_2^\star = F_4^\star)$$

$$\begin{array}{rcl}
F_1^* & = & f_1 \quad \wedge \\
F_2^* & = & \left( \begin{array}{lll} case & G_1^* = F_1^* & : f_1 \\ & TRUE & : f_2 \end{array} \right) \quad \wedge \\
F_3^* & = & \left( \begin{array}{lll} esac & & \\ case & G_1^* = G_2^* & : f_1 \\ & F_1^* = G_2^* & : f_2 \\ & TRUE & : f_3 \end{array} \right) \quad \wedge \\
consistent^E : & & \\
F_4^* & = & \left( \begin{array}{lll} esac & & \\ case & G_1^* = F_3^* & : f_1 \\ & F_1^* = F_3^* & : f_2 \\ & G_2^* = F_3^* & : f_3 \\ & TRUE & : f_4 \end{array} \right) \quad \wedge \\
G_1^* & = & g_1 \quad \wedge \\
G_2^* & = & \left( \begin{array}{lll} case & x_1 = x_2 & : g_1 \\ & TRUE & : g_2 \\ esac & & \end{array} \right)
\end{array}$$

and  $\varphi^E : consistent^E \rightarrow \varphi_\star^E$ .

# Appendix B

## Example of using Uninterpreted Functions: Translation validation

Uninterpreted Functions were used in the past both for property-based verification (proving that a certain property holds for a given model) and, more frequently, for proving equivalence between models. Many properties can be correct regardless of the exact functionality of a certain unit; in such cases the unit (which we assume is modelled as a function) can be replaced with an Uninterpreted Function.

As was previously written in this work, when proving equality between models, Uninterpreted Functions become even more useful. One example of this use, proving equivalence between two circuits, is shown in Chapter 1. Another example of this use is performing *Translation Validation*, a process of proving the semantic equivalence of input and output of a compiler. It is expected that every function in one side of the equation can be mapped to a similar function on the other side. In such cases replacing all functions with an uninterpreted version and using one of the reductions we saw in Sections A.1 and A.2 is typically sufficient for proving equivalence. The next example illustrates a Translation-Validation process which relies on Uninterpreted Functions and Ackermann's reduction. Unlike the hardware example, here we start from interpreted functions and replace them with Uninterpreted Functions.

Suppose that a source program contains a statement  $z := (x_1 + y_1) \cdot (x_2 + y_2)$  that the compiler that we wish to verify compiles into the following sequence of three assignments:

$$u_1 := x_1 + y_1; \quad u_2 := x_2 + y_2; \quad z := u_1 \cdot u_2$$



Note the two new auxiliary variables  $u_1$  and  $u_2$  that were added by the compiler. To verify this translation, we construct the verification condition

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 \cdot u_2 \rightarrow z = (x_1 + y_1) \cdot (x_2 + y_2)$$

whose validity we wish to check. For various technical reasons that are beyond the scope of this work, the verification condition is an implication rather than an equivalence<sup>1</sup>. For the purpose of demonstrating the use of Uninterpreted Functions this detail has no importance.

We now abstract the concrete functions appearing in the formula, such as addition and multiplication, by the abstract (uninterpreted) function symbols  $F$  and  $G$  respectively. The abstracted version of the above implication is

$$(u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2)) \rightarrow z = G(F(x_1, y_1), F(x_2, y_2))$$

Clearly, if the abstracted version is valid then so is the original concrete one.

Next, we perform the Ackermann reduction (see Definition A.1), replacing each functional term by a new variable but adding, for each pair of terms with the same function symbol, an extra antecedent which guarantees the functionality of these terms. Namely, that if the two arguments of the original terms were equal, then the terms should be equal.

Applying the Ackermann reduction to the abstracted formula, we obtain the following equality formula:

$$\begin{aligned} \varphi^E = & \left[ \begin{array}{l} (x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \rightarrow g_1 = g_2) \end{array} \right] \rightarrow \\ & ((u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1) \rightarrow z = g_2) \end{aligned} \quad (\text{B.1})$$

which we can rewrite as

$$\varphi^E = \left[ \begin{array}{l} (x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \rightarrow g_1 = g_2) \wedge \\ u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1 \end{array} \right] \rightarrow z = g_2 \quad (\text{B.2})$$

---

<sup>1</sup>Briefly, it means that we attempt to prove that the values allowed in the target code are also allowed in the source code, but not necessarily the other way. This asymmetry can be relevant when the source code is interpreted as a specification that allows multiple behaviors, only one of which is actually implemented. Such a relation is known as *refinement*.

The success of such a process depends on how different the two sides are. Suppose that we attempt to perform Translation Validation for a compiler that does not perform heavy arithmetic optimizations. In such a case the above scheme will probably succeed. If, on the other hand, we compare two *arbitrary* source codes, even if they are equal (they represent the same function) it is unlikely that we will be able to prove this fact following the same scheme. It is possible, for example, that one side uses the function  $2x$  while the other uses  $x + x$ . Since addition and multiplication are represented by two different Uninterpreted Functions, they will not be associated to each other in any way by Definition A.1, and hence the proof of equivalence will not be able to rely on the fact that they are semantically equal expressions.

# Appendix C

## Constructing smaller Equality Graphs

As was mentioned in Section 5.2, when Bryant’s reduction is used, all edges corresponding to comparisons between arguments of functions are ‘double’, meaning that they are both solid and dashed. In such a case, as the experiments show, RTC has no advantage at all, since every cycle is a Contradictory Cycle. In [RS01], Rodeh et al. suggested a method for constructing a much smaller Equality Graph from a given Equality Logic formula with Uninterpreted Functions. The method not only produces drastically smaller Equality Graphs, but also does not necessarily require a double edge for each comparison of function instances. The method changes the constructing of the Equality Graph. This improvement only apply to formulas that uses Bryant’s reduction. The construction of the graph is based on an analysis of the original equality formula with Uninterpreted Functions  $\varphi^{\text{UF}}$ , before the reduction to Equality logic. The main idea is to analyze which of the functional consistency constraints can potentially be used, and build the graph only according to them. Figure C.1 presents a graph that is built according to the Equality Logic formula that is generated by Bryant’s reduction, without the suggested optimization. In Figure C.2 the presented graph represents the same formula, but after the suggested optimization. Notice that this graph is much smaller and has no double edges. In fact, according to RTC, no transitivity constraints are required for this graph. The two figures and further explanation of the method can be find in [Str05].

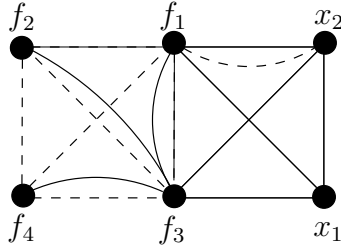


Figure C.1: Constructing the Equality Graph after the formula is reduced to Equality Logic. This results in a far more dense graph than the graph that is required by the optimization of [RS01](compare to Figure C.2).

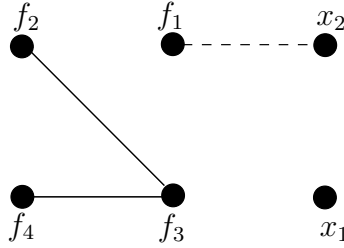


Figure C.2: The graph corresponding to the same original formula as C.1, but after applying the optimization of [RS01].

# Bibliography

- [Ack54] W. Ackermann. *Solvable cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
- [BGV99] R.E. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. 11<sup>th</sup> Intl. Conference on Computer Aided Verification (CAV'99)*, 1999.
- [BGV01] R.E. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, 2001.
- [BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. 14<sup>th</sup> Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.
- [Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
- [BV00] R.E. Bryant, , and M. Velev. Boolean satisfiability with transitivity constraints. In *Proc. 12<sup>th</sup> Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lect. Notes in Comp. Sci.*, 2000.
- [CLR00a] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 26, page 563. MIT press, 2000.
- [CLR00b] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 26, page 495. MIT press, 2000.

- [GS05] Roman Gershman and Ofer Strichman. Cost-effective hyper-resolution for pre-processing cnf formulas. In Toby Walsh and Fahiem Bacchus, editors, *Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.
- [GSZ<sup>+</sup>98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A.J. Hu and M.Y. Vardi, editors, *CAV98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
- [L.R04] L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
- [M81] Yannakakis M. Computing the minimum fill-in is np-complete. *SIAM Journal of Algebraic and Discrete Mathematics*, 2:77 – 79, 1981.
- [MMZ<sup>+</sup>01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
- [PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *Proc. 11<sup>th</sup> Intl. Conference on Computer Aided Verification (CAV'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999.
- [PRSS02] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and computation*, 178(1):279–293, October 2002.
- [Ros70] D. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32:597 – 609, 1970.
- [RS01] Y. Rodeh and O. Shtrichman. Finite instantiations in equivalence logic with uninterpreted functions. In *Computer Aided Verification (CAV)*, 2001.
- [Shi88] Y. Shibata. On the tree representation of chordal graphs. *J. Graph Theory*, 12:421–428, 1988.
- [Str05] Ofer Strichman. Lecture notes: Decision procedures for fragments of first-order logic. (unpublished), 2005.