TÉCNICO LISBOA

Inference in Biological Regulatory Networks

Alexandre Duarte de Almeida Lemos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Maria Inês Camarate de Campos Lynce de Faria
Professor Pedro Tiago Gonçalves Monteiro

Examination Committee

Chairperson: Professor António Manuel Ferreira Rito da Silva
Supervisor: Professor Pedro Tiago Gonçalves Monteiro
Member of the Committee: Professor Alexandre Paulo Lourenço Francisco

September 2016

Acknowledgments

First of all, I would like to thank my supervisors, Professor Inês Lynce and Professor Pedro Monteiro, for all their guidance, support and for always being there to help me solve my problems. This year, I have learned much about new topics, and it has been a real honour to be your master student. I am looking forward to work with you both again during my PhD.

I want to thank Professor Alexandre Francisco for all the comments and suggestions about my thesis.

I am also grateful to Miguel Neves for the help he gave me on how to use the *runsolver* tool on the server.

Thanks very much João Carraquico and João Luís for all your support during this last five years. It has been a pleasure working and discussing about everything with both of you.

I want to thank my family and all my friends for supporting me on this journey.

Finally, I would like to thank Fundação para a Ciência e a Tecnologia (FCT) for partially supporting this work through the research project DataStorm (EXCL/EEI-ESS/0257/2012).

Resumo

Atualmente, a maior parte dos modelos de sistemas biológicos ainda são elaborados manualmente, sendo propícios a erros humanos. Modelos diferentes podem ser obtidos a partir dos mesmos conjuntos de dados e provavelmente diferentes modeladores vão obter diferentes modelos. Sempre que se obtêm novos dados sobre um modelo é necessário confirmar se o modelo continua consistente. Se os novos dados estiverem incoerentes com o modelo é necessário corrigi-lo. As grandes quantidades de dados que estão envolvidas tornam este processo complicado para ser feito manualmente. Já existem algumas ferramentas para reparar modelos biológicos mas não são frequentemente utilizadas funções Booleanas para explicar a relação entre os componentes.

Neste trabalho são analisadas diferentes abordagens ao problema de modelar e reparar redes biológicas propondo um conjunto de reparações a aplicar no caso de redes inconsistentes. Este conjunto de reparações foi implementado em duas ferramentas usando uma ferramenta de satisfação máxima (MaxSAT) e programação de conjunto de resposta (ASP). Ambas as ferramentas permitem a reparação de redes biológicas descritas com um formalismo lógico e considerando múltiplos conjuntos de dados experimentais. Estas duas implementações foram testadas e comparadas usando duas redes biológicas da *Escherichia coli* e da *Candida albicans*. O conjunto de reparações conseguiu corrigir todos modelos sendo a versão de MaxSAT a mais eficiente a obter as soluções.

Palavras-chave: Redes regulação Booleanas, Reparação de modelos, Funções Booleanas, Programação de conjunto de resposta, Satisfação máxima

Abstract

Nowadays, most biological models are still handmade requiring a great amount of effort by the modeller. Different models can be derived from the same set of data and different modellers will therefore most likely build different models. Every time new data is obtained, it is necessary to reassess the model's consistency. If the model is not consistent with the new data, then it needs to be corrected. With the large amount of data available, repairing a model by hand is often a difficult task and so it is important to reduce the difficulty of this task by creating computational tools that allow the representation of models and to reason over them. There are already some tools that can repair models. However, these tools do not frequently allow the use of Boolean functions to explain the behaviour of the biological components.

In this work, different approaches to model and repair biological networks are analysed, proposing a new set of repair operations. This set was implemented using two different approaches, one using a Maximum Satisfiability (MaxSAT) solver and another using an Answer Set Programming (ASP) solver. Both tools can repair biological networks, using a logical formalism, with multiple sets of experimental data. The two tools were tested and compared using biological networks of *Escherichia coli* and *Candida albicans*. The set of repair operations proposed was able to solve all inconsistent models, and the solution implemented using Maximum Satisfiability (MaxSAT) was the most efficient.

Keywords: Boolean regulatory networks, Model repair, Boolean functions, Answer set programming, Maximum satisfiability

Contents

List of Tables

List of Figures

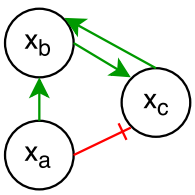
Chapter 1

Introduction

Nowadays, most biological models are still handmade and require a great amount of effort by the modeller. Different models can be derived from the same set of data and different modellers will therefore most likely build different models. Models of biological regulatory networks are increasingly used to formally describe and understand complex biological processes. Such models are often repaired whenever new observations become available, because the model cannot generate behaviours consistent with the new observations, or because the behaviours are contradictory. This process of model repair is often manual and therefore prone to errors. So, it is important to reduce the difficulty of this task by creating computational tools that allow the representation of models and to reason over them.

There are many possible formalisms [1, 2, 3, 4] to describe biological regulatory networks at a steady state. However, in this work focus on the logical formalism [1], specifically on the Boolean case. When considering this type of formalisms there are many tools one can use to reason over the network. Figure 1.1 shows a small network and the corresponding Boolean functions. The nodes *a*, *b* and *c* represent Boolean biological components and the functions K_i are used to explain the relationship between the values of the nodes. If one considers the experimental profile x_a=true, x_b=false and x_c=true one can see that the network is clearly inconsistent since the two function K_b and K_c are not coherent with the specified values. The work presented in this document proposes a set of repair operations that can correct these inconsistencies changing the two functions that originate them.

This type of problem falls within the scope of the general Boolean Satisfiability problem (SAT) [5], which is the problem of checking if a formula is satisfiable. This approach has many possible appli-



$$K_a(x) = x_a$$
$$K_b(x) = x_a \vee x_c$$
$$K_c(x) = x_b \wedge \neg x_a$$

Figure 1.1: The representation of small network (left) and the respective logical functions (right).

cations, from circuit verification to reasoning over biological networks (the application in focus here). Repairing a network can be seen as an optimization problem and so it can be encoded into Maximum Satisfiability (MaxSAT). MaxSAT is a generalization of Boolean Satisfiability (SAT) where the objective is to find an assignment that maximizes the number of clauses that are satisfied.

The Answer Set Programming (ASP) [6] language has the advantage that it is easy to write and does not require the development of complicated algorithms. This language is very similar to the Prolog language and in recent years it has evolved, becoming increasingly more efficient. One can encode biological network using ASP, which in principle is easier to write in comparison with the encoding required when using MaxSAT. In the other hand, using a MaxSAT solver should be faster to find an answer. Implementing two tools using both ASP and MaxSAT may help ensure a correct and coherent answer to the problem at hand.

Many works have been published using ASP [7, 8] and MaxSAT [9] to reason over biological networks. In this work, a new set of repair operations is proposed and it is implemented in both ASP and MaxSAT, allowing for some performance comparisons.

1.1 Motivation

Nowadays, the amount of biological data available is increasing at a very fast pace and it is a challenge to manage the complexity involved in the sheer amount of data that can be associated with a biological network. To manage this, one can use many types of computer models which will try to show the relation between the biological components in a less complex way, making it easier to reason over them. As previously said, there are already many ways one can describe a network with different degrees of complexity. Furthermore, as more data is available, the need to revise previously made models may arise. Repairing a network considering all the data sets available for that system is thus very important since new data about the system is continuously being generated. Work on this topic has already been published by different authors, especially using the Sign Consistency Model (SCM) [4]. However it is not common to find a set of repair operations when dealing with more detailed models, like when one is dealing with models encoded using a logical formalism. Considering the size of a normal biological regulatory network, revising a model may be very complicated and especially when many data sets are available at the same time. So it is important to have computational tools to help modellers revise networks. When repairing Boolean regulatory networks, one can repair the inconsistent functions associated with a specific node by choosing a correct function for that node. This type of repair has the particular challenge that there are many functions to choose from, since their number increases exponentially with the number of arguments, *i.e.* the number of regulatory components that interact with inconsistent nodes. . In this work, one of the objectives is to perform this type of repair while dealing with the combinatorial blow out.

1.2 Contributions

This document describes different ways of modelling real life systems, in particular how to model biological networks. Two different encoding methods (using MaxSAT and ASP) are used to model a biological network using a Boolean formalism. These encodings allow a user to check the satisfiability of the model considering all experimental data available for the network. Here, a set of repair operations is also proposed to repair inconsistent models making them coherent with all data available. The proposed repairs basically choose the correct Boolean function to describe the biological components relations. Both encodings can repair an inconsistent network and their performance is compared. Part of the work developed here, the ASP version, was accepted as a communication in INForum 2016 with the title "Repairing Boolean regulatory networks using Answer Set Programming" [10]. An extended version of the communication (the ASP version only) was published as a technical report [11].

1.3 Objectives

The study on how to model real life systems and how to reason over them is very important. Much work has already been developed in the area of modelling large and complex systems. When considering, for example, a biological system we have to take into consideration that it can be described correctly by different models. Every time new data is obtained, it is necessary to reassess the model consistency. If the model is not consistent with the new data, then it needs to be corrected. So, it is important to reduce the difficulty of this task by creating computational tools that allow the representation of models and provide ways to reason over them. The work described in this document also intends to propose a set of repair operations to apply to an inconsistent model to render it consistent. The large amount of data involved in these networks makes it important to build a tool that can reason over large networks in the shortest possible period of time. To this end two approaches are undertaken: modelling using ASP, and modelling using MaxSAT. Modelling using ASP is easier to modify but slower than using MaxSAT. It is also important to address the trade off between the two approaches.

1.4 Thesis Outline

The previous work found in the literature regarding the modelling of biological networks is the topic of chapter 2. The remainder of the chapter is focused on explaining the notions behind the tools that are commonly used in this field.

Chapter 3 is divided in three main parts. First, it is described and explained the different repair operations found in the literature for the different modelling formalisms discussed in the preceding chapter. The second part describes and explains the new proposed repairs to repair an inconsistent biological network. The last main section 3.3 explains the encoding needed to implement the above mentioned repairs in ASP and MaxSAT using an example from a real life data set describing the *Escherichia coli* response to a rapid increase in temperature.

Chapter 4 presents and discusses the results obtained when repairing real life instances of *Escherichia coli* and *Candida albicans* data sets, showing which repair is the best for these type of networks. The networks were repaired using both tools, allowing us to check which one is more efficient.

Chapter 2

Background

Here the fundamentals needed to understand this document are described. This chapter is divided in three main parts: the first where it is given an introduction to the different formalisms to model biological networks; the basic concepts of MaxSAT that will be used further on to encode networks; finally, an overview of ASP that will be also used to encode networks.

2.1 Biological networks

In this section different ways to model biological networks are addressed. These networks are composed by regulatory components, representing the expression levels of genes or the activity of their corresponding proteins. However, often the amount of available data detailing many biological processes is scarce and a qualitative (less detailed) model is more suited to describe them. Many qualitative mathematical formalisms exist [12], which have been applied to model biological networks, such as Petri nets [2], piecewise-linear differential equations [3], Sign Consistency Model (SCM) [4] or the logical formalism [1].

Petri nets constitute a well suited formalism to describe both quantitative and qualitative dynamical models. It has been successfully used to model biological regulatory networks, where *places* denote regulatory components and *transitions* denote regulatory interactions between components (see [2, 13] for further details).

Piecewise-linear differential equations were first published by Glass and Kauffman [3]; this approach provides a coarse-grained picture which is suited to describe the dynamics of regulatory networks, when little quantitative information is available which is the case for many networks of interest [14, 15].

The SCM expresses the relation between biological components through an influence graph, where the edges represent the interactions and the nodes represent the components. Each edge has a sign associated, $1/\text{-}1$ if an interaction is an activator/inhibitor, respectively. The value of a component is defined by the product of a regulator value with the sign of the corresponding interaction. Consequently, a component can be fixed at different values whenever different regulators allow it, rendering a given model too permissive to different experimental observations. Nonetheless, this was successfully used to reason over biological networks, in particular to reason over a network of *Escherichia coli*. These works

were implemented using Answer Set Programming (ASP) [7] and Maximum Satisfiability (MaxSAT) [9].

This work will consider the logical formalism, with a focus on the Boolean case, since it is possible to represent a multivalued network using only Boolean variables [16]. A network is composed by nodes, edges and functions. A Boolean logical regulatory graph is defined by:

- a set of *n* regulatory components $G = \{g_0, \cdots, g_n\}$, where each component is associated with a Boolean variable representing the level of expression or activity of the component;
- a set of edges *E*, where $(g_i, g_j) \in E$, with $i, j \in [1, \cdots, n]$ denotes a regulatory interaction between components g_i and g_j, *i.e.*, g_i is a regulator of (influences) g_j;
- to each component g_i there is an associated regulatory logical function, $K_i: B^k \rightarrow B$ where $B = \{0, 1\}$ and $k >= 0$, which defines its value based on the value of its regulators. Components without regulators are denoted as inputs and have constant values $\in B$.

In this case, a regulatory component is considered to be active/inactive if the value of the variable is $\mathrm{true/false}$. A regulatory logical function is defined by the combinations of three basic Boolean functions (AND, OR, NOT). The experimental profiles used here, describe the values of the nodes at a steady state of an experiment.

Biological networks can therefore be encoded into Conjunctive Normal Form (CNF). This encoding allows the use of logical based tools to reason over the networks. There are many tools that can be used, such as *Open-WBO* [17] (a MaxSAT solver) and *clasp* [18] (an ASP solver). To use them properly it is important to understand how they work.

2.2 Maximum Satisfiability

Maximum Satisfiability (MaxSAT) solvers can solve many problems like the travelling sales person [19], the debug of logic circuits [20] and also reasoning over biological networks [9]. Many algorithms that solve a MaxSAT problem apply a SAT solver iteratively. The SAT solvers are used as black-boxes, making it easy to change the SAT solver used (it can always be updated with new developments of SAT solvers).

In order to present the MaxSAT problem it is important to define first the concepts of SAT:

Definition 1. A literal, l, is either a Boolean variable x (positive literal) or its negation $\neg x$ (negative literal).

Definition 2. A clause c is a disjunction of *n* literals $c = l_1 \vee ... \vee l_{\mathrm{n}}$.

Definition 3. A formula in the Conjunctive Normal Form (CNF) is a conjunction of *n* clauses $\mathrm{f} = c_1 \wedge ... \wedge c_{\mathrm{n}}$. In this document, all formulas are written in CNF.

Definition 4. A partial (complete) assignment is a mapping between some (all) the variables of a formula and a Boolean value.

Definition 5. A positive (negative) literal is satisfied iff is assigned the value true (false). A clause is satisfied iff there is at least one literal satisfied. Otherwise the clause is unsatisfied. A formula is satisfiable iff there is at least one assignment where all the clauses are satisfied. A formula is unsatisfiable iff there is no assignment that makes the formula satisfiable.

Definition 6. SAT is the problem of checking the satisfiability of a Boolean formula, i.e., determining if a given formula has a complete assignment that satisfies all the clauses in the formula.

Most SAT algorithms require the formulas to be written in CNF. The formulas that are not represented in CNF can be transformed into CNF [21].

Considering the formula $\mathrm{f1}$,

$$\mathrm{f1} = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge x_3$$

which has three positive literals (x_1, x_2, x_3), and one negative literal ($\neg x_2$). This formula is satisfiable because there is at least one assignment where all the clauses are satisfied. In this case, a possible assignment that satisfies the formula is assigning x_1 as true, x_2 as false and x_3 as true.

We can use the resolution rule [22] to check the satisfiability/unsatisfiability of a given formula. The resolution rule is an inference rule that says that two clauses, one containing a literal and the other one its negation, imply a new clause, the resolvent, consisting of the other literals. In general,

$$\frac{\alpha \vee \beta_1 \vee ... \vee \beta_n \qquad \neg\alpha \vee \gamma_1 \vee .. \vee \gamma_n}{\beta_1 \vee .. \vee \beta_n \vee \gamma_1 \vee .. \vee \gamma_n}$$

where $\alpha, \neg\alpha$, all βs and all γs are literals and the line means consequence. In the specific case of both clauses having only one literal, α and $\neg\alpha$, the resolution rule will generate an empty clause.

Given the formula $\mathrm{f1}$, we can infer this new clause: $x_1 \vee x_3$ from the first two clauses.

The resolution rule can be use to check the satisfiability of a formula and it was originally used in the Davis-Putnam algorithm [23]. When the resolution rule can not be applied any more to the formula it is possible to conclude of the satisfiability of the original formula. If during the process an empty clause was generated the original formal is unsatisfiable, otherwise it is satisfiable. Nowadays, many of the most sophisticated SAT and MaxSAT solvers use Conflict-Driven Clause Learning (CDCL) algorithms that try to learn from the conflicts that may arise during the search for a satisfiable assignment. The basic version of these algorithms are explained in more detail in appendix A.

Finally, we can now introduce the MaxSAT problem.

Definition 7. The MaxSAT problem is a generalization of SAT, where the objective is to find an assignment that maximizes the number of satisfied clauses.

It is also possible to see the MaxSAT problem with a different perspective: finding an assignment that minimizes the number of unsatisfied clauses.

There are other variations of the MaxSAT problem and to address them we need to define a few more concepts.

Definition 8. In the partial MaxSAT problem the clauses have two different categories. Some clauses can be categorized as hard and the others as soft. The objective of partial MaxSAT is to find an assignment that satisfies all the hard clauses and maximizes the number of soft clauses that are satisfied.

Definition 9. A weighted clause is a clause that has associated a positive integer w. This integer represents the cost of not satisfying the corresponding clause.

Definition 10. In the weighted MaxSAT problem all the clauses $\mathrm{c_i}$ have a weight $\mathrm{w_i}$ associated. The goal is to minimize the sum of the costs of the clauses that are not satisfied.

Definition 11. In the weighted partial MaxSAT problem some clauses can be weighted as hard. To these hard clauses the same weight is assigned; the other clauses can have different weights as long as the sum of the weights of all the soft clauses is smaller than the weight of a single hard clause. The objective of weighted partial MaxSAT is to find an assignment that satisfies all the hard clauses and maximizes the weight of soft clauses satisfied.

The sum of the weight of all the soft clauses needs to be smaller than the weight of a single hard clause because otherwise it could cost less to satisfy all the soft clauses instead of a hard clause.

Definition 12. In the MaxSAT context the upper bound is the maximum cost that the unsatisfied clauses can have.The upper bound is used in some iterative algorithms to refine the solution.

There are many algorithms to solve the MaxSAT problem, that can be divided in two families, the iterative algorithms and the Core-guided algorithms. The *Open-WBO* [17] is one of the best MaxSAT tools which implements both unsatisfiability-based and linear search sat-unsat algorithms. The unsatisfiability-based uses a SAT solver iteratively and at each step the formula is relaxed until it is satisfiable. A linear search sat-unsat algorithm is another iterative approach to solve MaxSAT. This algorithm starts with a relaxed satisfiable version of the formula (to each clause is added a new variable) and at each iteration reduces the number of relaxed clauses until an unsatisfiable assignment is found. The optimal solution is the last satisfiable answer. This approach is also used in the tools like Sat4j library [24] and QMaxSAT [25]. A more detailed descriptions of the above mentioned algorithms are given in appendix B.

2.3 Answer Set Programming (ASP)

During this section, it is assumed that the reader has basic knowledge of SAT algorithms. If not please read the appendixes A and B in advance.

ASP is a form of declarative programming, using logic semantics to solve search problems. An ASP program is a set of rules and seems very similar to a Prolog program, in the sense that the rules in ASP are written in a similar way to the clauses in Prolog, making it easy to read. A program in ASP is the "definition" of the problem and when writing the program it is not necessary to create algorithms to solve it. However, the mechanisms to find a solution are different from Prolog, being a bit more similar to the ones used in some SAT solvers that will be described further on.

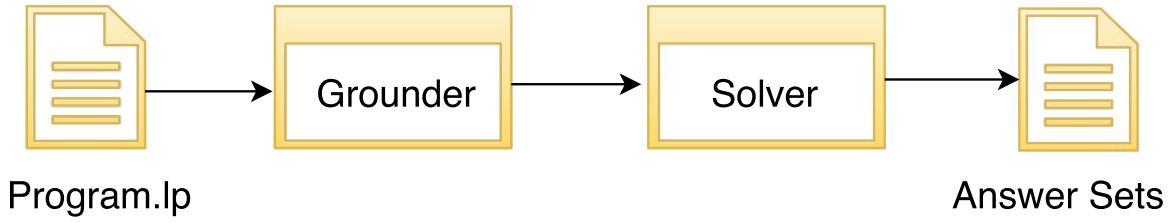


Figure 2.1: Solving process of an ASP program.

The DPLL algorithm (further explained appendix A) is mainly a backtrack algorithm and is behind many ASP solvers. This guarantees that a program will end, unlike a Prolog program where no such guarantee exists [26]. In a Prolog program the order of the rules is important to avoid infinite loops; in ASP the order of the rules is irrelevant. In order to understand the execution of an ASP program we have to explain two main parts: grounding and solving. Figure 2.1 shows the solving process, which starts with the grounder that receives the ASP program and transforms it into a variable free program (all possible rules are instantiated). The variable free program is the input of the solver which returns the answer sets (stable model) when possible for the initial program. Beforehand, we have to define the structure of an ASP program.

Definition 13. An ASP program P over a set of literals (*i.e.* a predicate in first-order logic) is a finite set of rules.

A rule r has a head and a body; it is written in following form

$$l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n$$

where l_i is a literal and $\sim l_i$ is its (default) negation. The left side of $\leftarrow$ is the head of the rule and so the head of r is

$$head(r) = \{l_0\}.$$

The right side is the body and the body of the rule r is

$$body(r) = \{l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n\}.$$

The body can be divided in two: $body(r)^+$ the positive literals and $body(r)^-$ the negative literals of the body. Considering the rule r:

$$body(r)^+ = \{l_1, ..., l_m\},$$

$$body(r)^- = \{\sim l_{m+1}, ..., \sim l_n\}.$$

The head is true if the body holds, i.e. if the positive literals, l_1 to l_m are true and the negative literals, $\sim l_{m+1}$ to $\sim l_n$ *can* be false [6]. The comma and the semi-colon represent the *and* and the *or* operation, respectively. All the rules in ASP can be written in this form; however sometimes there are simpler ways to write them (using fewer lines) as it will be seen further on.

$\leftarrow l$ is a rule without a head and thus represents a constraint and it means that l must not be satisfied.

A rule that only has a head, l, means that l must be satisfied and it is called a *fact*.

Definition 14. A set of literals is a model of the Program P if the set satisfies all the rules of P.

ASP has two types of negation, the classic negation ($\neg$) and the weak negation ($\sim$). The classic negation is not always the best representation for an open world, i.e. for this type of negation it is necessary to know that the literal is false for the negation to be true. In weak negation it is not necessary to know that the literal is false, it is something you can assume. For example, the rule

$$cross \leftarrow \sim train.$$

is using the weak negation and thus it is not appropriate for this case. It is dangerous to cross without knowing whether the train is coming. Using the classic negation is better in this case:

$$cross \leftarrow \neg train.$$

The weak negation allows more flexibility and the use of nonmonotonic reasoning; on the other hand some models may become unstable. In the case of having a program,

$$l \leftarrow \sim l$$

it means that if we do not have l it is possible to infer l and so this program does not have any stable models. If that program was written using the classic logic there would be a model, l.

Definition 15. Let P be a program. For any set M of literals from P, let $\mathrm{P_M}$ be the program obtained by deleting:

i. each rule that has a negative literal $\sim B$ in its body with $B \in M$,

ii. all negative literals in the bodies of the remaining rules.

Clearly, $\mathrm{P_M}$ is negation-free, so that $\mathrm{P_M}$ has a unique minimal model[1]. If this model coincides with M, then we say that M is a stable model of P [27].

Consider now these two rules [28]

$$p_1 \leftarrow p_1.$$
$$p_2 \leftarrow \sim p_1.$$

Let $M = \{p_1\}$. Then the second rule is removed by case *i* of definition 15. The minimal model that satisfies the first rule is $\{\}$ and so different from M. Hence it is not a stable model. Now let consider that $M = \{p_2\}$. Now only $\sim p_1$ is removed (case *ii* of definition 15) and in this case M coincides with the model. Hence $\{p_2\}$ is a stable model. There are still two possible values for M, $M = \{p_1,\ p_2\}$ and $M = \{\}$. In the first case the second rule is removed exactly as in the first case considered and again it

[1] The subset minimal model that satisfies $\mathrm{P_M}$.

is not stable. Now let $M = \{\}$. In this case, only $\sim p_1$ is removed but the resulting model is different from M and so not stable.

In ASP, as in Prolog and in first order logic, it is possible to expresses predicates. $p(l_0, ..., l_n)$ represents a predicate p with n arguments. This predicate can be used as a part of the body or as a head of a rule. For example, it can be used to represent a fact, such as the current colour of a traffic light:

$$colour(s, red).$$

In ASP there are different types of rules that simplify the writing of the program. Examples include:

- cardinality constraints;
- multiple choice;
- weighted rules;
- aggregation rules;
- optimization statements;
- conditional literals;

.

It is possible to express cardinality rules by writing them in this form:

$$l_0 \leftarrow n\{l_1, ..., l_m\}.$$

This means that the head belongs to a stable model if at least *n* literals of the body are satisfied. Another possible constraint rule

$$l_0 \leftarrow \{l_1, ..., l_m\}n.$$

states that at most *n* literals need to be satisfied. These two constraints can be used simultaneously.

An ASP program accepts multiple choices, unlike a Prolog program, and it would be written in this form:

$$\{l_0, l_1\} \leftarrow l_2.$$

If l_2 is satisfied then l_0 AND/OR l_1 can be added to the stable model. The multiple choice can be bound with at least and at most constraints.

A weighted rule has the following form:

$$l_0 \leftarrow b\{l_1 = w_1, ..., l_m = w_1\}u.$$

and the head is added to a stable model if $b \leq \sum_{i=1}^{m} l_i w_i \leq u$ is satisfied. It is possible to associate a few aggregation functions to this weighted rule. For example, it is possible to perform the sum of the weights or to compute the maximum/minimum weight by adding a key word *sum/max/min* just after b.

Concerning the simplification of the rules, ASP allows us to write using conditional literals. For example, the fact that the traffic light on the street *s* can either be green, yellow or red may be expressed by:

$$1\{colour(s, green),\ colour(s, yellow),\ colour(s, red)\}1 \leftarrow streetlight(s).$$

or by a shorter version of the same facts:

$$1\{colour(s, c) : colour(c)\}1 \leftarrow streetlight(s).$$

assuming that the program has the following facts $colour(green)$, $colour(yellow)$ and $colour(red)$.

This simplification can also be used in the body of a rule and for optimization statements. An optimization statement causes the solver to minimize/maximize the sum of the weights when computing the optimal stable model. Minimization statements are written in this form

$$minimize\{l_1 = w_1@p_1, ..., l_n = w_n@p_n\}.$$

where l_i is a (positive or negative) literal with an associated weight w_i and a priority level p_i. The priority level can be used when combining optimization statements in order to express which is more important. A maximization statement can be represented as minimization with the symmetric weights. For example, the maximization statement where the literal l_i has weight w_i it is possible to be written by minimization statement instead, giving the literal l_i the weight $-w_i$.

Definition 16. A rule is safe if all its variables exist as positive literals in the body of some rule. If all the rules are safe then the program is safe.

Definition 17. An instance of a rule is a rule where all the variables were replaced by elements in the Herbrand universe [29] of the program (a universe that contains all the constants from the program and every function whose arguments belong to this universe).

The grounders (explained later on), like gringo [30], need to receive a safe program otherwise the grounding process will not guarantee termination.

Grounding

Grounding is used to compute a finite and succinct representation of a program. The union of all instances of the rules in P is the grounding of the program P. This basic implementation causes many unnecessary rules because the union of all instances of the rules includes all the combinations of elements of the domain even if some of these rules can never be derived.

Another way of looking to the grounding is proceeding in a bottom-up fashion where at each iteration the domain will be bigger. This avoids the problem of generating unnecessary rules because at each step it will only generate the rules that are possible. It starts with the *facts* and after each iteration it will add the new *facts*, the ones that were inferred. The big disadvantage of this procedure is that

at each iteration all the rules are again processed. The grounding tools dlv[2] [31] or gringo[3] [30] try to avoid reprocessing the rules. In order to avoid reprocessing, semi-naive evaluation (based on lazy evaluation) is applied. This means that the system will evaluate the rules based on the *facts* discovered until the last iteration is reached. Imagine that a *fact* p is discovered, everything from now on will be evaluated considering this discovery. Rules that need this *fact* to be true, will be simplified by removing this condition from the body.

Consider

$$p(a, b).$$
$$p_1(X) \leftarrow p(X, Y).$$

Without using a smart grounding the previous program would generate $p_1\ (a) \leftarrow p(a, a)$, $p_1\ (a) \leftarrow p(a, b)$, $p_1\ (b) \leftarrow p(b, a)$ and $p_1(b) \leftarrow p(b, b)$ even knowing that only the second rule can be reached. In a bottom-up version grounding would start with the *facts*, in this case $p(a, b)$ and so only $p_1(a) \leftarrow p(a, b)$ would be generated. In the next iteration the algorithm would try to generate rules based on $p(a, b)$ and $p_1(a)$. In this case, the algorithm would end given that there are no more possible rules to generate.

Solving

ASP solvers make use of SAT solvers, formerly DPLL-like SAT solvers and more recently CDCL-like SAT solvers (explained on appendix A). In order to make a SAT solver work in context of an ASP program, the CDCL algorithm is changed to use the nogoods concept.

Definition 18. A nogood is a set of literals that if satisfied violates a constraint.

This concept can be used for performing inference in ASP as unit propagation[4]. For example the implication of two literals $l_2 \leftarrow l_1$ can be viewed as a disjunction $\neg l_1 \vee l_2$ and this has a nogood set corresponding to the assignment of l_1 as true and l_2 as false.

The solving algorithm computes the stable model (in case one exists) and learns the nogoods from the conflicts that arise, a process which has the same objective as learning new clauses in the CDCL algorithm. During the propagation it is necessary to consider the possibility of loops in the formulas because their existence can make the solving process infeasible [32]. However, it is possible to check their satisfiability.

A Strongly Connected Component (SCC) (representing the loop between rules) can be detected by analysing a dependency graph of the program. It is possible to discover nogoods from the cycles in order to avoid an infeasible problem. To discover these nogoods it is necessary to compute the unfounded set. An unfounded set [33] of a program with an incomplete assignment is a set containing the head of every rule of the program where either:

[2]dlv system is available from: http://www.dlvsystem.com/
[3]gringo system is available from: http://potassco.sourceforge.net/
[4]Unit propagation is a procedure that iteratively satisfies the unit clauses.

- the body of the rule is not satisfied by the partial assignment;
- the intersection of the $body(rule)^+$ and the unfounded set is not empty.

Consider the following program with a loop:

$$l_1 \leftarrow l_2$$
$$l_2 \leftarrow l_1$$

Consider the partial assignment where l_1 is assigned the value true. In this case, l_2 is the unfounded set.

Consider an arbitrary literal $a \in unfoundedSet$; the new nogood is the result of the union of a assigned true with all the literals of the external body of the $unfoundedSet$ assigned true. Consider U as a set of literal of the ASP program. The external body of U is the body of all the rules where the head of the rules belong to U and the intersection of the $body(r)^+$ and U is empty ($body(\{r \in Program \mid head(r) \in U,\ U \wedge body(r)^+ = \emptyset\})$) [6, 32]. In this case, l_2 is a nogood.

Chapter 3

Inference in Biological Regulatory Networks

One important aspect of reasoning over biological networks is to be able to check the satisfiability of a model. In this case a network is considered to be consistent if and only if all nodes of the network are consistent. A node is considered to be consistent if it is an input node or if its value corresponds to the value given by the Boolean function that explains it.

It is normal that a model may require revision after new experimental data about the modelled system is available. In the literature there are many different types of possible operations to change a network in order to make it consistent.

3.1 Background on repairs

The study of biological regulatory networks normally follows two approaches: inferring a model (the nodes, edges and functions) from experimental data [34] or revising an existing model based on conditionally data [9]. The first approach tries to automate the process of creating a model to explain the data. The second approach focuses on repairing an existing model based on new data that has arisen. When considering the logical formalism, specifically the Boolean case, one needs to choose the correct function to represent the interactions between the regulators of a given regulatory component. When revising a model is possible to apply a similar process, re-choosing the correct function to replace a previously inconsistent function (*i.e.* a function that is incoherent with the data available).

When considering logical models, even after inferring a model from experimental data, it is important to be able to verify whether the model can replicate the experimental behaviour [35]. So, using the logic formalism is important to revise the model, changing the Boolean function (*e.g.* changing an AND function into an OR function).

Another problem that arises when revising a model is how to choose the correct repair, *i.e.* how to choose the ”best” repair among the possible repairs. It is possible to choose it by minimizing the cardinality or the subset but it is also possible to use a few more empirical types of choices. Merhej

et al [36] propose a few rules of thumb to serve as guidelines to choose possible repairs of a network. For example, taking into account the diameter of the network (shortest path between two nodes that are furtherest apart), one may define an interval for the acceptable diameter for a given network. Using this interval one can assign a cost to a given repair, and so instead of choosing the cardinality minimal repair (least changes to the model) it will be chosen the repair that follows the empirical rules previously defined.

In the following, an overview of the literature regarding the repair of Biological networks considering different modelling formalism will be presented.

Revising a model is a difficult task since it involves the creation of possible repair pathways that must make sense in the biological field and that are feasible to implement.

Logic Formalism Mobilia *et al.* [37] proposed an approach where there are two types of constraints: the additivity and the observability constraints. The observability constraints are the union of inequalities of the kinetic parameters (the expression levels of a gene). Additivity constraints are used to indicate that generally no inhibition (activation) can occur in case of an edge with a positive (negative) sign. In this case, repairing a model is changing the additivity constraints so that they are coherent with observability constraints (from the experimental data). Mobilia *et al.* proposed to remove constraints additivity as possible repair operation to an inconsistent model. The authors note that some of the possible removable constraints suggested by the program would not be biologically plausible. In references [38] and [39] the authors also propose removing weaker constrains as possible repair operations. Sometimes it is important to further explore the model, trying to infer new interactions or components. Nevertheless, if a new interaction is inferred, then it is important to re-test the consistency of the model in order to remove other constraints for which the consistency test fails.

Sign Consistency Model One approach used to model and repair biological regulatory networks is the Sign Consistency Model (SCM) [4]. When one is using the SCM a network is said to be consistent iff all nodes are consistent. A node is consistent if at least one regulator explains the sign of the node, *i.e.* if the product of a regulator value with the sign of the corresponding interaction is coherent with the value of the node. In the literature, when trying to repair a network using the SCM [9, 7, 40, 8] it is normal to include several repair operations. These operations can focus on repairing the model or correcting the data. When one considers the correction of the model it is possible to perform the following operations:

- Flipping the sign of an edge, that in biological terms means that a node in reality is not the activator/inhibitor that was originally defined in the model but rather the reverse.
- Adding new edges are normally used when considering incomplete models. However, it is a complex repair since it has the potential risk of adding an arbitrary number of edges which can enlarge the encoding of the network.
- Turning a node into an input node, meaning that this node is consist no matter what.

When considering the repair of the data, it is also possible to change the sign of the nodes [7].

Formalism	Repair	Source
SCM	make input, add edge, flip edge and node	[8, 7, 9, 40]
Extended SCM	SCEN-FIT, Minimal Correction Sets (MCoS)) OPT-SUBGRAPH, OPT-GRAP	[41, 42]
Markov logic	thumb rules used as cost for adding/removing edges	[36]
Differential Equations	change the structure, adjust parameters	[44, 43]
Boolean	change a Boolean function	[35]
Logic	remove constraints	[38, 39, 37]

Table 3.1: Summary of repairs found in the literature.

Extended Sign Consistency Model Melas *et al.* [41] suggest four possible repair operations to remove inconsistencies from an extended version of the SCM, adding the possibility of a sign being *0* (e.g. when a node has a positive and negative influence) and not only *1* or *-1*. The authors classify the problem in two types, one considering a single experiment and the other considering multiple experiments. When having a single experiment, *SCEN-FIT* tries to minimize the number of inconsistencies between the experimental data and the model. The Minimal Correction Set procedure is used to correct the inconsistencies found before by adding a minimal number of external influences to the model. When having multiple experiments, the authors propose two repairs: *OPT_SUBGRAPH* that tries to find a subgraph (removing edges) that minimizes the inconsistencies in all experiments; and *OPT_GRAPH* that removes edges and at the same time adds new influences to the model.

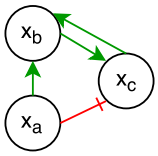
Thiele *et al* [42] also apply the Minimal Correction Set procedure to a further extended SCM. They propose an edge can have two more signs, i.e. instead of using only $+$ and $-$ they use $+, -, \ominus$ and $\oplus$. These two additions make use of the weak negation of ASP since they represent a possibility. The symbol $\ominus$ ($\oplus$) represents an interaction that clearly is not an activation (inhibition) but it may not be an inhibition (activation).

Piecewise-linear differential equations When considering piecewise-linear differential equations there are other types of repairs. In reference [43], the authors divide the revision process in two parts: one where they change the structure of the user-made model to better fit the data and another one where they try to find the best parameters for a particular type of regulation (casual effect). Note that not all combinations of structures are possible since some of them may not have biological sense.

Table 3.1 shows a summary of the repair operations available on the literature organized by types of formalism.

3.2 Proposed solution

This work focuses on checking the satisfiability of a Boolean regulatory network and proposing, in case of inconsistency, possible repair operations. This work uses the Boolean logical formalism as the way to



$$K_a(x) = x_a \qquad K_a(x) = x_a$$

$$K_b(x) = x_a \vee x_c \qquad K_b(x) = \neg(x_a \vee x_c)$$

$$K_c(x) = x_b \wedge \neg x_a \qquad K_c(x) = \neg(x_b \wedge \neg x_a)$$

Figure 3.1: The representation of small network (left), the respective logical functions (center) and the repaired logical functions for an experimental profile x_a=true, x_b=false and x_c=true (right).

model biological the networks. As we are using the Boolean formalism, it is possible to focus the repair operations on changing the inconsistent functions. When considering this form of repair, it is important to change the function for a similar function that satisfies all data available for the network. In this formalism, changing a function is a repair operation that covers many other different repair operations used in the literature. For example, a commonly used repair when considering the SCM is changing the sign of an edge; the equivalent repairs can be used in this formalism: complementing the value of a function's argument.

Figure 3.1 shows a network (left) with the corresponding Boolean functions (center). A regulator is represented by a green arrow and the negation of a regulator is represented by a red hammerhead arrow. The edge *(a,c)* is the only edge representing a negation of a regulator on the network showed in Figure 3.1 and so the corresponding Boolean rule has x_a negated.

After creating a model, such as the one described in Figure 3.1, new observations of the process under study may arise that should also be explained by the model. Combining both may generate inconsistencies and so the model will need to be revised in order to find a new model that also satisfies the new data. For example, considering the experimental profile: x_a=true, x_b=false and x_c=true, one can see that the model in Figure 3.1 is inconsistent, since the functions that explain the value of the node b and c are incoherent with the experimental profile. The potential model revisions should try to find repairs that make the model coherent to all the available data.

Here, we propose four atomic repair operations to the logical functions, which can be further combined:

e - removes a regulator (ensuring that a component has at least one regulator, *i.e.*, never becomes an input);

i - negates any number of regulators of a function;

n - changes an AND/OR function into a NAND/NOR function, respectively;

g - changes an AND into an OR, a NOT into the $\mathrm{identity}$ function and vice verse.

Considering the model described in Figure 3.1 and the given experimental profile, the Boolean functions (center) can be repaired by applying two repairs of type *n*, *i.e.* by negating the functions K_b and K_c (Figure 3.1 right). These repairs are cardinality minimal, corresponding to the minimal number of

Input		Functions															
A	B	A	B	$\neg A$	$\neg B$	$A \wedge B$	$\neg A \wedge B$	$A \wedge \neg B$	$\neg A \wedge \neg B$	$A \vee B$	$\neg A \vee B$	$A \vee \neg B$	$\neg A \vee \neg B$	$A \oplus B$	$A \equiv B$	T	F
0	0	0	0	1	1	0	0	0	1	0	1	1	1	0	1	1	0
0	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0	1	0
1	0	1	0	0	1	0	0	1	0	1	0	1	1	1	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1	1	0	0	1	1	0

Table 3.2: All possible combinations of Boolean functions with two arguments.

repairs required to correct the model. However, it is possible to perform other types of repairs to correct this model, such as removing the NOT and negating the remaining regulators of the inconsistent functions. Applying the repair *gi* can perform the above mention solution. The resulting functions would be $K_b(x) = \neg x_a \vee \neg x_c$ and $K_c(x) = \neg x_b \wedge x_a$ which are consist with the experimental profile.

Also, when one combines the repair *i*, which allows the negation of any number of regulators of a function (only allows the negation of regulators that have not been previously negated), with the repair *g*, the output will be more general than when applying only the repair *n*. The functions NAND and NOR are a subgroup of the functions produced when combining those repairs. Consider the OR function $(a \vee b)$, using repair *n* it is possible to obtain the NOR function $\neg(a \vee b)$. De Morgan's laws say that $\neg(a \vee b)$ is equal to $(\neg a \wedge \neg b)$ and this last function can be obtained by repair *gi*.

Generically, the number of possible Boolean functions that can be used to repair a function will increase with the number of regulators of a component. For example, the number of possible functions with two arguments is sixteen and this number will increase exponentially with the number of arguments (number of regulatory components that influence one specific component). The growth will follow the expression 2^{2^n} where n is the number of arguments of the function [45]. Table 3.2 presents all the possible combinations of functions with two arguments. Consider the function K_c described in the network shown in Figure 3.1. Now using the following experimental profiles p_0 .. p_3:

$$p_0 : \ x_a{=}\text{true}, \ x_b{=}\text{false} \text{ and } x_c{=}\text{true};$$

$$p_1 : \ x_a{=}\text{false}, \ x_b{=}\text{true} \text{ and } x_c{=}\text{false};$$

$$p_2 : \ x_a{=}\text{false}, \ x_b{=}\text{false} \text{ and } x_c{=}\text{true};$$

$$p_3 : \ x_a{=}\text{true}, \ x_b{=}\text{true} \text{ and } x_c{=}\text{false};$$

one can see that the function K_c makes the network inconsistent. K_c is a function with two arguments and so any of the sixteen functions shown in Table 3.2 can be a possible solution. In this case, only one function fits in the four experimental profiles. In order to find the function that fits this data it is necessary to remove one of its arguments. Column $\neg B$ of Table 3.2 represents the only function that has the correct values x_c (the result of the function) for the arguments x_a (column A) and x_b (column B). To conclude, only function $\neg B$ can correctly represent the relations between the nodes. At the end, node x_a does not influence x_c.

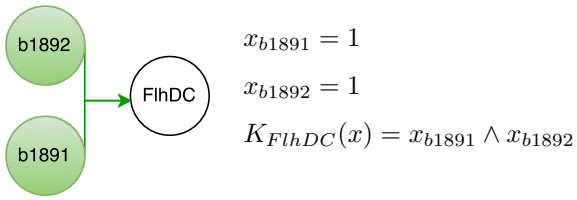


Figure 3.2: The representation of a small part of the network described in HeatShock data set (left) and the respective logical functions (right). A node in green/red represents the assignment of the value $\mathrm{true/false}$ to the regulatory component by the experimental profile. Uncoloured nodes corresponds to unassigned components.

Considering the case of a function with two arguments, by combining repairs *e*, *i* and *g*, one can achieve the total of twelve functions (all basic Boolean functions, plus one of the derived Boolean functions, the implication). The functions XOR, NXOR (equivalence operation), true and false are not achievable by these repairs. Table 3.3 shows the different combinations of repairs needed to convert the binary function $\neg A \wedge B$ into a different one (whenever possible). For example, according to the four experimental profiles shown above K_c should be $\neg x_b$ instead of $\neg x_a \wedge x_b$. In order to convert a function of this type $\neg A \wedge B$ to $\neg B$ one needs to use the combined repair *ei* ($\neg B$ column).

The Boolean formalism used in this work allows the model to have any type of Boolean functions. The function XOR may be defined in a model to explain the behaviour of a given node by the creation of temporary nodes, combining the three basic functions $(A \wedge \neg B) \vee (\neg A \wedge B)$. However, with the repair operations described above the XOR function will not be considered a possible correction to an inconsistent function since all these repairs are based on a single function and not on the combination of different functions. If for some reason the original model already has the necessary temporary nodes to define an XOR function, it is possible, even with these repairs, to obtain it.

3.3 Implementation

In this section, it is explained the implementation of two tools to repair biological networks using the formalism shown in the previous chapter. The first tool was implemented using ASP and the second was encoded using Partial MaxSAT. These two tools produce exactly the same result, the only differences between them being the difficulty of adding new functionalities and the execution time.

	$A \wedge B$	$A \wedge \neg B$	$\neg A \wedge \neg B$	$\neg A \vee B$	$A \vee B$	$A \vee \neg B$	$\neg A \vee \neg B$	B	A	$\neg B$	$\neg A$
repair	g	g,i	i	g	g	g, i	g, i	e	e,g	e,i	e

Table 3.3: Possible repairs for the function ¬ A ∧ B and which repairs are used to achieve them.

3.3.1 Encoding into Answer Set Programming

To encode the network shown in Figure 3.2, it is necessary to write the predicate $\mathrm{node(v)}$ for each node v and the predicate $\mathrm{obs\_vlabel(P,V,S)}$ to give to each node an observed value S in an experimental profile P. P on the predicate $\mathrm{obs\_vlabel(P,V,S)}$ allows us to handle multiple sets of observations independently at the same time. The predicate $\mathrm{edge(v_1,v_2)}$ represents an unidirectional relation between two nodes $\mathrm{v_1}$ and $\mathrm{v_2}$. Four types of basic Boolean functions can be encoded - AND, OR, $\mathrm{identity}$ and NOT - through the predicate $\mathrm{func<F>(N,O)}$, where $<\mathrm{F}>$ is the name of the function, N a unique identifier and O the output node of the function $<\mathrm{F}>$. The unique identifier N is used to set the arbitrary number of arguments of the function. The predicate $\mathrm{regulator(N,V)}$ associates the function with the number N with the argument V. To define functions with more than one argument, it is necessary to define one predicate for each argument. It is possible to construct more complex functions, using temporary nodes to combine functions. The encoding for the network shown in Figure 3.2 is shown in Figure 3.3. The predicate exp defines the name of an experimental profile. The predicate temp is used to define temporary nodes used to create more complexed functions.

```
funcAnd(1,FlhDC).    edge(b1892,FlhDC).                edge(b1891,FlhDC).
regulator(1,b1892).  obs_vlabel("heatShock",b1892,1).  obs_vlabel("heatShock",b1892,1).
regulator(1,b1891).  exp("heatShock").                 node(FlhDC).
node(b1892).         node(b1891).
```

Figure 3.3: The ASP encoding for the network shown in Figure 3.2.

$$\mathrm{consistentFunc(P,O)} \leftarrow \mathrm{funcAnd(N,O),\ noneNegative(O,N,P),\ vlabel(P,O,1),\ \sim repair(funcOr(N,O)).} \quad (1)$$

$$\mathrm{consistentFunc(P,O)} \leftarrow \mathrm{funcAnd(N,O),\sim noneNegative(O,N,P),\ vlabel(P,O,0),\ \sim repair(funcOr(N,O)).} \quad (2)$$

$$\mathrm{noneNegative(V,N,P)} \leftarrow \sim \mathrm{oneNegative(V,N,P),\ onePositive(V,N,P).} \quad (3)$$

$$\mathrm{pos(funcNand(N,O))} \leftarrow \mathrm{repair_n,\ funcAnd(N,O),\ \sim isNandNor(O),\ \sim repair(funcOr(N,O)).} \quad (4)$$

$$\mathrm{pos(rEdge(U,V))} \leftarrow \mathrm{repair_e,\ edge(U,V),\ edge(W,V),\ W!{=}V,\ W!{=}U,U!{=}V,\ \sim rEdge(U,V),\ \sim rEdge(W,V).} \quad (5)$$

$$\mathrm{pos(regulator(N,V))} \leftarrow \mathrm{repair_i, \sim isRegulatorNot(N,V),\ regulator(N,V).} \quad (6)$$

$$\mathrm{pos(funcAnd(N,O))} \leftarrow \mathrm{repair_g,\ funcOr(N,O).} \quad (7)$$

$$\leftarrow \mathrm{vlabel(P,V,S),} \sim \mathrm{input(P,V),} \sim \mathrm{consistentFunc(P,V).} \quad (8)$$

Figure 3.4: Part of the ASP encoding to check the consistency and repair the network.

A predicate $\mathrm{consistentFunc(P,V)}$, where P is a profile and V is a node, is generated if and only if the value of the node V is coherent with the result of the function that explains the presence of this node in the experimental profile P. The rules (1) and (2) are used to verify the consistency of an AND function if the function AND is not repaired and so replaced by an OR. The predicates $\mathrm{vlabel(P,O,S)}$ and

$\mathrm{obs\_vlabel(P,O,S)}$ are similar, since both represent the value of the node V in the experimental profile P. $\mathrm{vlabel(P,O,S)}$ is inferred when the observed value ($\mathrm{obs\_vlabel(P,O,S)}$) is present or, if no observed value was specified, with a previously computed value. The predicate $\mathrm{noneNegative}$ is inferred (3) when all the regulators of the node have the value true.

The consistency check of the network, with or without repairs, is made with two auxiliary predicates ($\mathrm{onePositive}$ and $\mathrm{oneNegative}$). The presence of these predicates indicates that a node V influenced by the function N and based on the profile P has at least one regulator with the value $\mathrm{true/false}$.

The non-existence of an instance of the predicate $\mathrm{consistentFunc(P,V)}$ for a non-input node in a given profile means that the network is inconsistent (8). Note that all nodes without incoming edges are considered input nodes.

Rule (4) shows the possibility of creating a NAND function; to use this repair it is necessary to have the repair active ($repair_n$) and the existence of an AND function (not previously repaired to an OR/NAND/NOR).

Removing an edge is a possible repair when the flag $repair_e$ is active and there are at least two edges that were not removed for the given node (rule 5).

Rules (6) and (7) are used to infer the possibility of applying a repair *i* or *g*. Rule (6) ensures that a previously negated regulator is not negated more than once.

Note that the repair operations changes the structure of the model for all experimental profiles, and so the repair operations need to be able to make the model coherent with all experimental profiles available.

The complete encoding is available at $\mathrm{http://web.ist.utl.pt/\sim alexandre.lemos/rbnasp/}$.

3.3.2 Encoding into Maximum Satisfiability

In order to use a MaxSAT solver, the biological network has to be encoded into CNF and for that it is necessary to define a few Boolean variables. For each node v, there is a Boolean variable $label_v$ such that $label_v$ is assigned the value $\mathrm{true/false}$ if node v is active/inactive. Every edge between two nodes can have a function NOT associated. In order to describe it two auxiliary variables are used:

- $f\mathrm{NOT}_{vu}$ is a Boolean variable that when assigned true represents the existence of the function NOT on the edge between the nodes v and u;
- $rf\mathrm{NOT}_{vu}$ is a Boolean variable that represents the result of the application of the NOT function on the edge between the nodes v and u.

With these variables it is possible to write constraints to allow a MaxSAT solver to check the network's consistency. First, for each node a unit clause [1] corresponding to the value of the node on the experimental profile is created. There will also be created unit clauses for the variables $f\mathrm{NOT}_{vu}$. These clauses will consist of a single positive literal. When one only wants to check the satisfiability of a network it is not necessary to have a list of unit clauses for every edge without a NOT function. To encode the function one can use the Tseitin transformation [21] that creates a CNF formula based on an arbitrary circuit. A function AND is encoded as follows:

[1] A clause that has only one unassigned literal and so its assignment is necessary for the clause to be satisfied.

$(\neg r_0 \vee .. \vee \neg r_n \vee v) \wedge (r_0 \vee \neg v) \wedge .. \wedge (r_n \vee \neg v)$,

where r_0 and r_n are regulators and v is the output of the function. Very similarly one can encode an OR function:

$(r_0 \vee .. \vee r_n \vee \neg v) \wedge (\neg r_0 \vee v) \wedge .. \wedge (\neg r_n \vee v)$.

The encoding for the NOT function is the following:

$(\neg r \vee \neg v) \wedge (r \vee v)$.

This encoding requires that the number of clauses for each function is the number of regulators plus one. Note that it is not necessary to have a variable to identify an input node since an input is not regulated and so the solver will not check its consistency.

With this type of encodings it is possible to verify the consistency of the network shown in Figure 3.2 with the following CNF formula:

$$(b1891) \wedge (b1892) \wedge (b1891 \vee \neg FlhDC) \wedge (b1892 \vee \neg FlhDC) \wedge (\neg b1891 \vee \neg b1892 \vee FlhDC)$$

The encoding has five clauses where two of them are unit clauses for the value of the nodes *b1892* and *b1891*. The other three clauses represent the Boolean function AND and it is easy to see that there is a satisfiable assignment for this example, assigning *FlhDC* to true.

Repairing a network is an optimization problem that can be encoded into Partial MaxSAT. The files containing the encoding are generated specifically to each repair operation, *i.e.* each file only has the encoding that can be reached. For example, when it is only possible to remove edges, the encoding does not need to support the possible negation of a function. This allows to have shorter encodings when possible. To apply the repairs previously shown it is necessary to add more constraints depending on which repair is available.

Repair *e* In order to remove an edge, it is necessary to add a new variable, N_{vu} for each edge (v, u) that can be removed. This variable is used to deactivate clauses, *i.e.* satisfy clauses where an edge is removed. When N_{vu} is assigned the value true it means that the edge from u to v has been removed. With this repair one could remove all edges going to a node. To avoid removing all the edges into a node a clause $(\neg N_{vu} \vee .. \vee \neg N_{wu})$ is defined, ensuring that each node has at least one incoming edge. The variable N_{vu} is added to all clauses containing the variables for the node v and u at the same time. This repair only influences the functions with more than one argument. As it is possible to see, the encoding of these functions has two types of clauses: one type where only one regulator is present, and another type where we have all the regulators together. When the variable N_{vu} is assigned to true, the clauses that contain this variable are trivially solved. However, the second type of clauses also becomes true under this condition. To solve this problem it is necessary to replicate this clause with all combinations of regulators. These new clauses are trivially solved when the variable N_{vu} is assigned to false. A unit clause for the variable N_{vu} will be created. This clause will consist of a single negative literal and will be considered a soft clause.

For example, the AND function with two arguments is now encoded as follows:

$(N_{r0}v \vee \neg r_0 \vee N_{r1}v \vee \neg r_1 \vee v) \wedge (N_{r0}v \vee r_0 \vee \neg v) \wedge (N_{r1}v \vee r_1 \vee \neg v)$.

Repair *g* To allow a function to be changed, for example from AND to OR, it is necessary to use variables to describe which function is associated to each node. For each regulatory logical function K_v there is a variable $f{<}function{>}_v$ that is assigned the value $\mathrm{true/false}$ if the function for the node v is of the type $<function>$. In this case, $<function>$ can only be AND or OR. It is only specified the value for the active function (otherwise changing a function would cost twice). This value is represented as a soft unit clause. This way this constraint can be broken when needed. All nodes have encoded the OR an AND functions. The clauses required to encode the functions OR and AND are the same as shown before. However, these clauses need to have one more variable. Each clause of the encoding of the function needs to include the literal $\neg f{<}function{>}_v$ in order to specify which function is currently active. For example, the AND function with two arguments is now encoded as follows:

$(\neg f\mathsf{AND}_v \vee \neg r_0 \vee \neg r_1 \vee v) \wedge (\neg f\mathsf{AND}_v \vee r_0 \vee \neg v) \wedge (\neg f\mathsf{AND}_v \vee r_1 \vee \neg v)$.

In order to have only one type of function for each node (*e.g.* having the AND and OR function for the node v) it is necessary to define the following constraint:

$$\bigwedge_{v \in V} \bigvee_{f, f_1 \in \{f\mathsf{AND}, f\mathsf{OR}\}} \neg f_v \neg f_{1v}. \tag{9}$$

This constraint guarantees that at most one function is active for a given node. However, this is not enough since we want to have exactly one function per node. To achieve this we need to add a new constraint:

$$\bigwedge_{v \in V} f\mathsf{OR}_v \vee f\mathsf{AND}_v. \tag{10}$$

This repair also allows changing a NOT function into an IDENTITY function. For this it is necessary to encode the IDENTITY function. The IDENTITY function is encoded like the AND or OR function but with only one argument, $(fnot_{vu} \vee v \vee \neg u) \wedge (fnot_{vu} \vee \neg v \vee u)$. The unit clause representing the assignment of $fnot_{vu}$ as true, *i.e.* $(fnot_{vu})$, is considered a soft clause.

Repair *i* In a similar way as the repair *g* deals with the change between AND and OR, this repair deals with having a function's argument negated or not. In order to do that it is necessary to have a variable that is assigned to true when the argument is negated and false otherwise. As said before, this repair can only negate an argument that has not been previously negated. So all the arguments that are negated on the model are described like when considering repair *g* with the exception that now the unit clause $(f\mathsf{NOT}_v)$ is considered hard instead of soft since these regulators are already negated. The constraints that are described for repair *g* (about the function NOT and IDENTITY) are instantiated for all arguments. Now, for the arguments that are not already negated, the variable $f\mathsf{NOT}_v$ is assigned as false in a soft clause making it possible to negate a function's argument.

Repair *n* For this repair, it is necessary to add two more variables to describe the functions NAND and NOR for each node. When generating the encoding for this repair alone, it is only necessary to

have one of these variables for each node since it is impossible to change an AND for an OR. To verify the consistency of these new functions it is necessary to define some more constraints. These new constraints are trivially solved if the corresponding function is inactive, *i.e.* if the variable $f\langle\text{function}\rangle_v$ is false. To describe the behaviour of the NAND function it is required to:

$(\neg r_0 \vee .. \vee \neg r_n \vee \neg v \vee \neg f\text{NAND}_v) \wedge (r_0 \vee v \vee \neg f\text{NAND}_v) \wedge .. \wedge (r_n \vee v \vee \neg f\text{NAND}_v)$.

The encoding for the NOR is:

$(r_0 \vee .. \vee r_n \vee v \vee \neg f\text{NOR}_v) \wedge (\neg r_0 \vee \neg v \vee \neg f\text{NOR}_v) \wedge .. \wedge (\neg r_n \vee \neg v \vee \neg f\text{NOR}_v)$.

The encoding previously shown for the AND and OR functions requires the addition of the negative literal corresponding to the function in each clause. The constraints (9) and (10) are needed but considering only the reachable functions with the active repairs (*e.g.* when considering AND as the function for node v and the repair *n* as the only repair possible NAND is the only reachable functions).

Combining repairs When combining repairs the encoding is basically the instantiation of the combination of the encoding discussed above, with two more little details:

- First, when considering a combination of repairs including both repairs *g* and *n* it is required to change the domain of the constraints (9) and (10) given that all functions are reachable.
- Finally, it is harder to change a function from an AND function to a NOR function than an AND function to an OR function. These repairs should have a higher cost and so a new variable is created for the following constraints: $(\neg f\text{NOR}_v / f\text{NAND}_v \vee aux_v) \wedge (f\text{NOR}_v / f\text{NAND}_v \vee \neg aux_v) \wedge (\neg aux_v)$. The first two constraints are soft and so, like in the ASP version, the cost is two. The cost of transforming an AND (OR) function into a NOR (NAND) function will be two since two soft clauses cannot be satisfied.

The summary of the clauses and weights (soft and hard) necessary to guarantee the correct functioning of the different repair operations is shown in Table 3.4. The encoding required to define the three basic functions (AND, OR, NOT) in order to allow the use of the different combinations of repairs is shown in Table 3.5. Note that for example when considering repair *i* it is necessary to add temporary variables, *i.e.* the edge (r_i, v) will become two edges (r_i, t_i) and (t_i, v). Finally Table 3.6 shows the summary of the encoding required when the active repair operations allows one function to be negated (NAND, NOR). All the clauses presented in the last two tables (describing the encoding for the Boolean functions) are weighted as hard.

Repair	Soft clauses	Hard clauses
Repair *e*	$(\neg N_{vu})$	$(\neg N_{vu} \vee .. \vee \neg N_{wu})$
Repair *i*	if $K_v(x)! = \neg x_u$ then $(f\text{NOT}_v)$	if $K_v(x) = \neg x_u$ then $(\neg f\text{NOT}_v)$
Repair *g*	if $K_v(x) = \neg x_u$ then $(\neg f\text{NOT}_v)$ if $K_v(x) = x_u \wedge x_w$ then $(\neg f\text{AND}_v)$ if $K_v(x) = x_u \vee x_w$ then $(\neg f\text{OR}_v)$	$(f\text{OR}_v \vee f\text{AND}_v)$
Repair *n*	if $K_v(x) = x_u \wedge x_w$ then $(\neg f\text{AND}_v)$ if $K_v(x) = x_u \vee x_w$ then $(\neg f\text{OR}_v)$	$(f\text{OR}_v / f\text{AND}_v \vee f\text{NOR}_v / f\text{NAND}_v)$

Repair *gn*	if $K_v(x) = \neg x_u$ then $(\neg f\text{NOT}_v)$
	if $K_v(x) = x_u \wedge x_w$ then
	$(\neg f\text{AND}_v)$
	$(\neg f\text{NOR}_v \vee aux_v) \wedge (f\text{NOR}_v \vee \neg aux_v)$
	if $K_v(x) = x_u \vee x_w$ then
	$(\neg f\text{OR}_v)$
	$(\neg f\text{NAND}_v \vee aux_v) \wedge (f\text{NAND}_v \vee \neg aux_v)$
		$(f\text{OR}_v \vee f\text{AND}_v \vee f\text{NOR}_v \vee f\text{NAND}_v)$
		$(\neg aux_v)$

Repair *gi*	if $K_v(x) = x_u \wedge x_w$ then $(\neg f\text{AND}_v)$ if $K_v(x) = x_u \vee x_w$then $(\neg f\text{OR}_v)$ else $(f\text{NOT}_v)$	$(f\text{OR}_v \vee f\text{AND}_v)$

Table 3.4: Summary of the constraints required in the MaxSAT encoding depending on the active repair operations. The combinations of repairs not presented are obtained simply by adding the previous encondings. $f<type>_v$ is the variable indicating the type of Boolean function (AND, OR, NOR, NAND, NOT) for node v. The variables $N_{r_0 v}$ to $N_{r_n v}$ represent the possible removal of the edge (r_i, v).

Function	Repairs	Clauses
AND	Without	$(\neg r_0 \vee .. \vee \neg r_n \vee v) \wedge (r_0 \vee \neg v) \wedge .. \wedge (r_n \vee \neg v)$
	n, *g*, *gi*, *in gin*	$(\neg f\text{AND}_v \vee \neg r_0 \vee .. \vee \neg r_n \vee v) \wedge (\neg f\text{AND}_v \vee r_0 \vee \neg v) \wedge .. \wedge (\neg f\text{AND}_v \vee r_n \vee \neg v)$
	e, *ei*	$(N_{r_0v} \vee \neg r_0 \vee .. \vee \neg r_n \vee N_{r_nv} \vee v) \wedge (\neg N_{r_0v} \vee .. \vee \neg r_n \vee N_{r_nv} \vee v) \wedge .. \wedge (N_{r_0v} \vee \neg r_0 \vee .. \vee \neg N_{r_nv} \vee v)$[2] $\wedge (N_{r_0v} \vee r_0 \vee \neg v) \wedge .. \wedge (N_{r_nv} \vee r_n \vee \neg v)$
	en, *eg*, *ein*, *egi*, *egin*	$(\neg f\text{AND}_v \vee N_{r_0v} \vee \neg r_0 \vee .. \vee \neg r_n \vee N_{r_nv} \vee v) \wedge (\neg f\text{AND}_v \vee \neg N_{r_0v} \vee .. \vee \neg r_n \vee N_{r_nv} \vee v) \wedge .. \wedge (\neg f\text{AND}_v \vee N_{r_0v} \vee \neg r_0 \vee .. \vee \neg N_{r_nv} \vee v)^2 \wedge (\neg f\text{AND}_v \vee N_{r_0v} \vee r_0 \vee \neg v) \wedge .. \wedge (\neg f\text{AND}_v \vee N_{r_nv} \vee r_n \vee \neg v)$
OR	Without	$(r_0 \vee .. \vee r_n \vee \neg v) \wedge (\neg r_0 \vee v) \wedge .. \wedge (\neg r_n \vee v)$.
	n, *g*, *gi*, *in*, *gin*	$(\neg f\text{OR}_v \vee r_0 \vee .. \vee r_n \vee \neg v) \wedge (\neg f\text{OR}_v \vee \neg r_0 \vee v) \wedge .. \wedge (\neg f\text{OR}_v \vee \neg r_n \vee v)$
	e, *ei*	$(N_{r_0v} \vee r_0 \vee .. \vee r_n \vee N_{r_nv} \vee \neg v) \wedge (\neg N_{r_0v} \vee .. \vee r_n \vee N_{r_nv} \vee \neg v) \wedge .. \wedge (N_{r_0v} \vee r_0 \vee .. \vee \neg N_{r_nv} \vee \neg v)^2 \wedge (\neg r_0 \vee v \vee N_{r_0v}) \wedge .. \wedge (\neg r_n \vee v \vee N_{r_nv})$
	en, *eg*, *ein*, *egi*, *egin*	$(\neg f\text{OR}_v \vee N_{r_0v} \vee r_0 \vee .. \vee r_n \vee N_{r_nv} \vee \neg v) \wedge (\neg f\text{OR}_v \vee \neg N_{r_0v} \vee .. \vee r_n \vee N_{r_nv} \vee \neg v) \wedge .. \wedge (\neg f\text{OR}_v \vee N_{r_0v} \vee r_0 \vee .. \vee \neg N_{r_nv} \vee \neg v)^2 \wedge (\neg f\text{OR}_v \vee \neg r_0 \vee v \vee N_{r_0v}) \wedge .. \wedge (\neg f\text{OR}_v \vee \neg r_n \vee v \vee N_{r_nv})$
NOT	Without, *e*, *n*, *en*	$(\neg r \vee \neg v) \wedge (r \vee v)$
	i, *in*, *ei*, *ein*	if $K_v(x) != \neg x_u$ then $\{(\neg f\text{NOT}_v \vee \neg r \vee \neg v) \wedge (\neg f\text{NOT}_v \vee r \vee v) (f\text{NOT}_{vu} \vee v \vee \neg u) \wedge (f\text{NOT}_{vu} \vee \neg v \vee u)\}$
	g, *gn*, *eg*, *egn*	if $K_v(x) = \neg x_u$ then $\{(\neg f\text{NOT}_v \vee \neg r \vee \neg v) \wedge (\neg f\text{NOT}_v \vee r \vee v) (f\text{NOT}_{vu} \vee v \vee \neg u) \wedge (f\text{NOT}_{vu} \vee \neg v \vee u)\}$
	egin	$(\neg f\text{NOT}_v \vee \neg r \vee \neg v) \wedge (\neg f\text{NOT}_v \vee r \vee v) (f\text{NOT}_{vu} \vee v \vee \neg u) \wedge (f\text{NOT}_{vu} \vee \neg v \vee u)$

Table 3.5: Summary of the MaxSAT enconding for the three atomic functions (AND, OR, NOT) depending on the active repair operations. The variables r_0 to r_n represent the arguments of the function and v its output. $f<type>_v$ is the variable indicating the type of Boolean function (AND, OR, NAND, NOR, NOT) for node v. The variables N_{r_0v} to N_{r_nv} represent the possible removal of the edge (r_i, v).

[2]Each clause represents a possible combination of regulators.

Function	Repairs	Clauses
NOR	*n*, *in*, *gin*	$(r_0 \vee .. \vee r_n \vee v \vee \neg f\text{NOR}_v) \wedge (\neg r_0 \vee \neg v \vee \neg f\text{NOR}_v) \wedge .. \wedge (\neg r_n \vee \neg v \vee \neg f\text{NOR}_v)$
	en, *in*, *egn*, *ein*, *egin*	$(N_{r_0 v} \vee r_0 \vee .. \vee r_n \vee N_{r_n v} \vee v \vee \neg f\text{NOR}_v) \wedge (N_{r_0 v} \vee r_0 \vee .. \vee \neg N_{r_n v} \vee v \vee \neg f\text{NOR}_v) \wedge .. \wedge (\neg N_{r_0 v} \vee .. \vee r_n \vee N_{r_n v} \vee v \vee \neg f\text{NOR}_v)$[3] $\wedge (\neg r_0 \vee \neg v \vee \neg f\text{NOR}_v \vee N_{r_0 v}) \wedge .. \wedge (\neg r_n \vee \neg v \vee \neg f\text{NOR}_v \vee N_{r_n v})$
NAND	*n*, *in*, *gin*	$(\neg r_0 \vee .. \vee \neg r_n \vee \neg v \vee \neg f\text{NAND}_v) \wedge (r_0 \vee v \vee \neg f\text{NAND}_v) \wedge .. \wedge (r_n \vee v \vee \neg f\text{NAND}_v)$
	en, *in*, *egn*, *ein*, *egin*	$(\neg r_0 \vee N_{r_0 v} \vee .. \vee \neg r_n \vee N_{r_n v} \vee \neg v \vee \neg f\text{NAND}_v) \wedge (\neg N_{r_0 v} \vee .. \vee N_{r_n v} \vee \neg r_n \vee \neg v \vee \neg f\text{NAND}_v) \wedge .. \wedge (\neg r_0 \vee N_{r_0 v} \vee .. \vee \neg N_{r_n v} \vee \neg v \vee \neg f\text{NAND}_v)$[3] $\wedge (r_0 \vee v \vee \neg f\text{NAND}_v \vee N_{r_0 v}) \wedge .. \wedge (r_n \vee v \vee \neg f\text{NAND}_v \vee N_{r_n v})$

Table 3.6: Summary of the MaxSAT enconding for the NAND and NOR functions depending on the active repair operations. The variables r_0 to r_n represent the arguments of the function and v its output. $f{<}type{>}_v$ is the variable indicating the type of Boolean function (AND, OR, NAND, NOR, NOT) for node v. The variables $N_{r_0 v}$ to $N_{r_n v}$ represent the possible removal of the edge (r_i, v).

[3]Each clause represents a possible combination of regulators.

Chapter 4

Results

In order to verify if the proposed repairs are sufficient to repair biological networks, an evaluation was run using biological data of *Escherichia coli* and *Candida albicans*. The evaluation was also intended to compare the ASP solution with the MaxSAT solution.

4.1 Experimental Setup

The network of *Escherichia coli* was obtained from $\mathrm{RegulonDB}$ [46]. The data sets HeatShock [47] and *Stat vs Exp* [48] were used for evaluating the proposed approach. The HeatShock data set describes the *Escherichia coli* response to an increase of temperature. The *Stat vs Exp* data set corresponds to the Exponential-Stationary growth shift study of *Escherichia coli*. These data sets were obtained from Gebser *et al* [7]. Apart from the complete data set (100%), subsets containing only part of the data were also used: 3%, 6%, 9%, 12%, 15%. For the partial data sets (3%, 6%, 9%, 12%, 15%), there are 200 different files with a randomly picked sample of the data. For the complete data set, there is only one file since the file includes all the observations of the corresponding data set. Using these different percentages one can understand better the behaviour of the different implemented tools considering different amount of data. It is easier to satisfy the experimental data in a 3% profile rather than in a 100% profile since it has less constraints. The evolution of the repairs can also be studied based on the different results obtained for each percentage. Since the models were imported from the SCM, and this formalism does not have a Boolean function associated to each node, the models were adapted considering that all nodes have the same (default) Boolean function. In order to better understand the behaviour of the regulatory components described in these data sets, they were tested considering two different default functions, AND and OR, combining the set of regulators. This means that for each data set, we consider two models: one where all the nodes with regulators have the AND function, and another where all the nodes with regulators have the OR function.

The tests were executed using the *runsolver* tool [49] with a time out of 600 seconds and a limit of 3 Gb of memory. The ASP program was executed using *Gringo* (version 4.5.4) [30] and *Clasp* (version 3.1.4) [18]. The MaxSAT encoding was executed using the MaxSAT solver *Open-WBO* (version

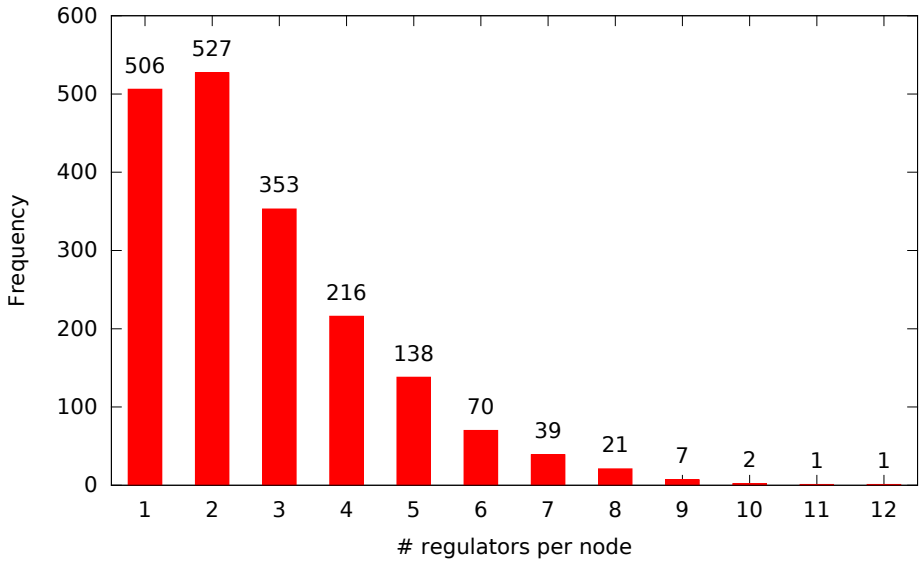


Figure 4.1: Distribution of the number of regulators per node for *Escherichia coli* model.

1.3.1) [17]. The two different implementations were run on a computer running *Ubuntu 14* equipped with 24 CPUs at 2.6 GHz and 64 Gb of RAM.

It is interesting to see that for the *Escherichia coli* model most nodes have a small number of regulators, limiting the search space of possible functions. This information is shown in Figure 4.1. This network has 1915 nodes, not considering temporary nodes that are required in the encodings (for negating regulators and functions), of which 34 are considered input nodes since they have no incoming edges. The encodings used require the additional nodes achieving the grand total of around 3200 nodes. The network starts with 1881 default functions and 1327 NOT functions.

The regulatory network for *Candida albicans* is larger but still most nodes have a small number of regulators, as it i shown in Figure 4.2. This network has 6410 nodes not considering temporary nodes that are required in the encodings (for negating regulators and functions), of which 71 are considered input nodes since they have no incoming edges. The encodings require the use of temporary nodes, achieving the grand total of around of 17000 nodes. The network starts with 6339 default functions and 10774 NOT functions.

It is important to note that the encoding for repair *e* is shorter for a function with a small number of arguments. For example when considering the CNF encoding shown in the Chapter 3 the number of new clauses required for a function with $\#\mathrm{args}$ arguments is given by this formula:

$$2 * \#args + \sum_{i=1}^{\#args} {}^{\#args}C_i, \ where \ {}^{\#args}C_i = \frac{\#args!}{i!(\#args - i)!}. \tag{4.1}$$

Since the nodes with one regulator are also connected through the default function, the repair *g* (which allows changing from AND to OR and vice-versa) does not change the output of the function. Moreover, these unary functions have only one possible change, through being negated, either by negating the default function creating a NAND or a NOR (repair *n*), or by negating the only regulator (repair *i*). Considering unary functions as a class of its own may increase the performance, making it possible to change

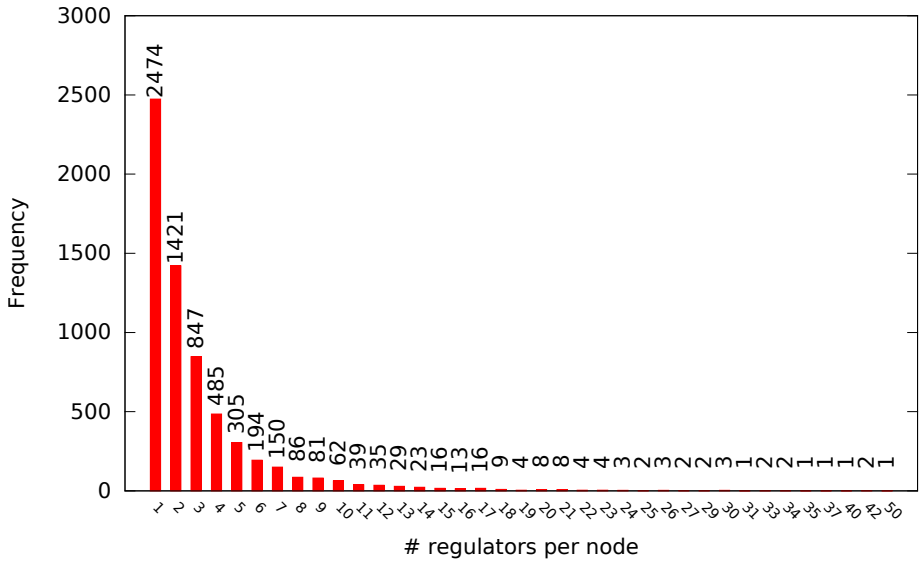


Figure 4.2: Distribution of the number of regulators per node for *Candida albicans* model.

the unary function to a NOT function.

The tests were run to find the cardinality minimal repair, although sometimes due to the time limit and memory limit it was not possible to find an optimal solution.

4.2 Stat vs Exp case study

Figure 4.3 shows the smallest number of identified repairs needed to correct the model (not necessarily cardinality minimal) for both default functions obtained using the ASP approach. Note that Figure 4.3 does not include the combinations of repairs that do not find a solution. All the functions covered by the repairs that find a solution are a superset of the set of functions covered by the repair *n*. In most cases, when considering the OR function as the default function it is necessary to apply less repairs. When considering the complete Stat vs Exp data set one can see that the repair *n* is the optimal solution for this case. It is the only one that achieves an optimal solution within the time limit. The other three types of basic repairs are not able to find any feasible solution.

Repair *n*, contrary to other repair types that find feasible solutions, has the smallest function coverage (smaller number of possible repairs to try) and, as such, it finds the optimal solution faster, if it exists. It provides a good solution for nodes with the value true that have influencing nodes with different values. These nodes need to be corrected when the default function is an AND function, but are consistent when considering the OR function as the default function. The data set has some nodes with the value false when the influencing nodes are different. Conversely, these nodes are consistent when running with the AND function but need correction with the OR function. Since the false nodes are less common then the true nodes, using the OR function as default function requires less repairs. These networks have also many nodes with the value true when their regulators are all false which can be repaired by negating the default function.

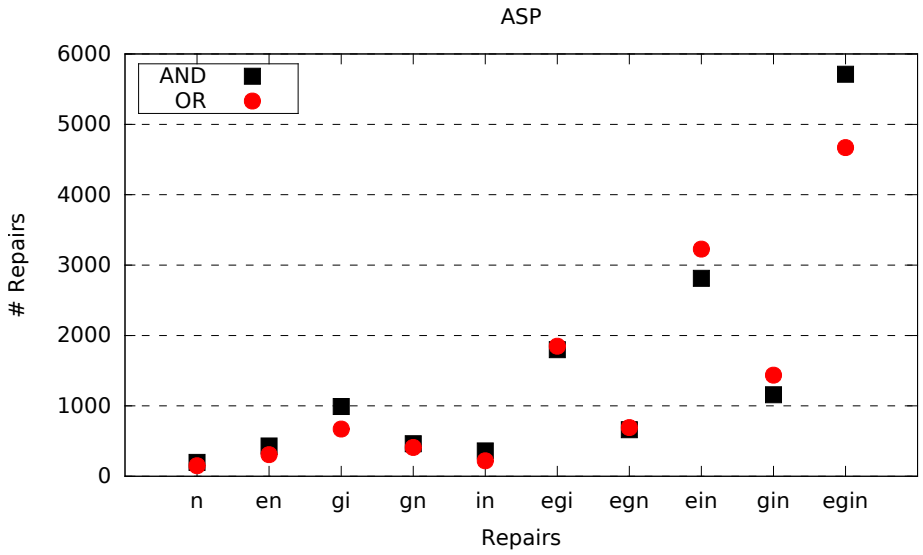


Figure 4.3: Number of repairs necessary to correct the Stat vs Exp data set. The types of repairs that are not present did not find a feasible solution. Repair *n* is the only one to find an optimal solution. The results shown were obtained using ASP.

All the combinations of repairs that find a solution include the functions covered by repair *n*. Hence, if the tests are run without constraints, then they should find a solution at least as good as the solution found by repair *n*. The repairs that were closer to the number of repairs obtained by repair *n* and with the default function AND, were run again with a time out of 3600 seconds. Figure 4.4 shows the minimal number of repairs that are needed to correct the model. However, none of these attempts reached an optimal solution. On average, the reduction in the number of repairs was sixteen, and none of the repairs came close to the number of repairs required by repair *n*.

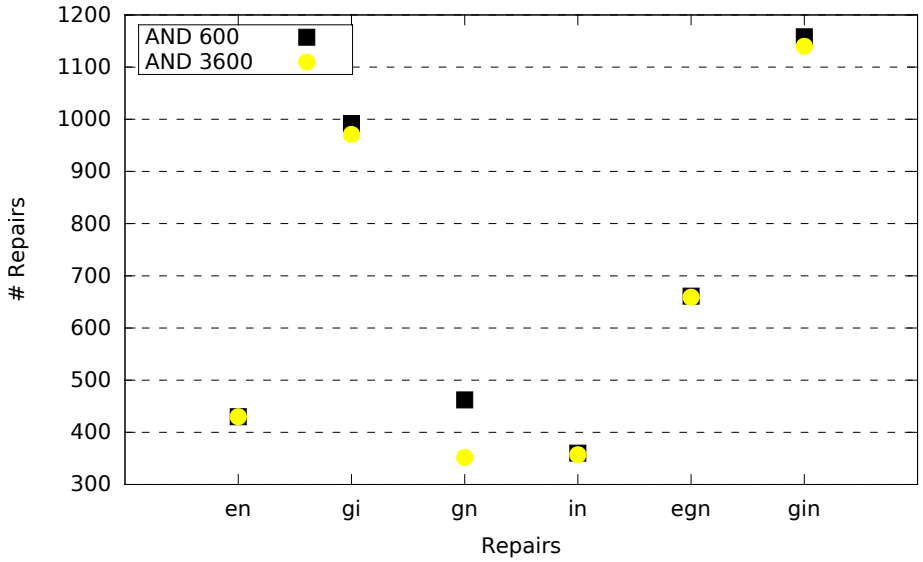


Figure 4.4: The number of repairs necessary to correct the Stat vs Exp data set for the repairs *en, gi, gn, in, egn* and *gin*, when running the tests with a time limit of 600 and 3600 seconds. The results shown were obtained using ASP.

Repair *n* is the only repair that obtained the optimal solution. In an attempt to optimize the solution

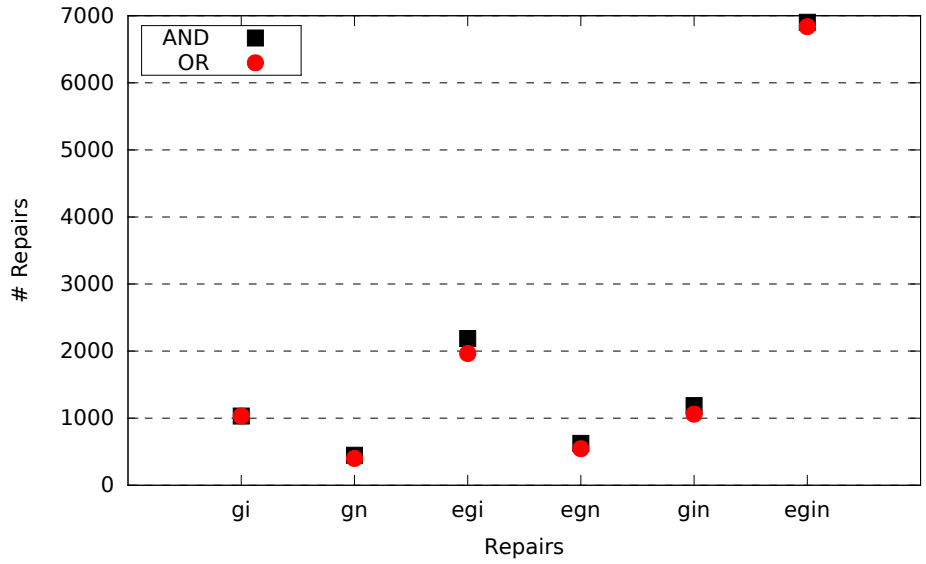


Figure 4.5: Number of repairs necessary to correct the Stat vs Exp data set, when considering two default functions (AND/OR and identity). The types of repairs that are not present did not find a feasible solution. All the repairs are feasible but not optimal. The results were obtained using ASP.

available in the repairs containing the repair *n*, an upper bound was added. The upper bound was the number of repairs needed to repair the model using only repair *n*. This reduces the search space, but with the time out of 600 seconds no solution was obtained.

Running the same tests for the whole data using the optimization previously explained (the use of an identity function for unary operation), one can see a few differences. First, there are no optimal solutions mainly because repair *n* is no longer valid. The disappearance of this solution is easy to explain through the fact that it is necessary to correct IDENTITY functions, replacing them with a NOT function, where

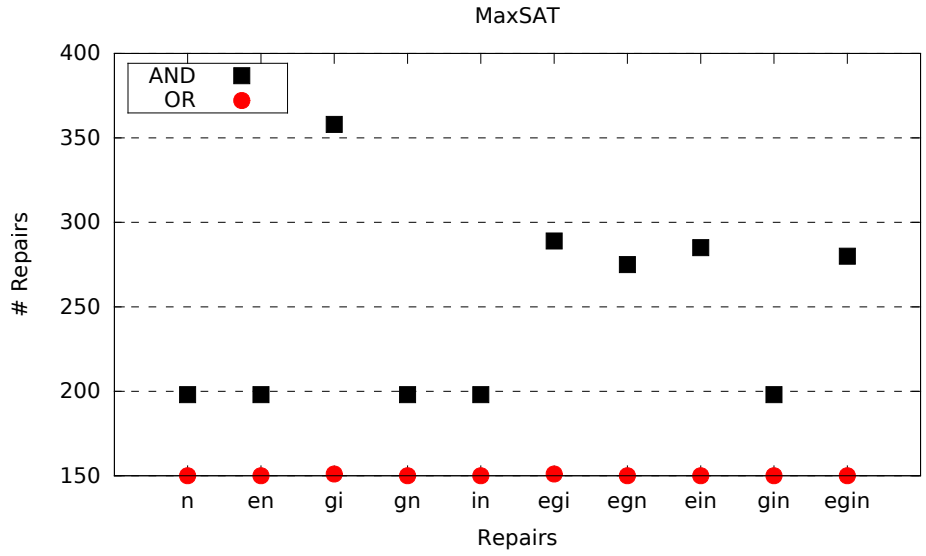


Figure 4.6: Number of repairs necessary to correct the Stat vs Exp data set. The types of repairs that are not present did not find a feasible solution. In this case only repairs used with the default function AND did not find a optimal solution. In particular repairs *gi*, *egi*, *egn*, *ein* and *egin*. The results were obtained using MaxSAT.

previously they were transformed from AND/OR into NAND/NOR. Here, only the repairs containing the repair *g* are possible. The results obtained in this test are presented in Figure 4.5. The difference in the required number of repairs between the AND function and the OR function as default functions is minimal.

When running exactly the same tests, *i.e.* the data from *Stat vs Exp* data set, with the limit of 600 seconds but using the MaxSAT solution are shown in Figure 4.6. The number of repairs needed is smaller since in most cases the program is able to find an optimal solution. Even when an optimal solution is not found the number of repairs is smaller. Considering the repair operations that find an optimal solution one can see that it is easier to repair the model considering the OR function as the default function.

AND	198 (O)	198 (O)	358	198 (O)	198 (O)	289	275	285	198 (O)	280
OR	150 (O)	150 (O)	151 (O)	150 (O)	150 (O)	151 (O)	150 (O)	150 (O)	150 (O)	150 (O)
	n	en	gi	gn	in	egi	egn	ein	gin	egin

Table 4.1: Number of repairs necessary to correct the Stat vs Exp data set with MaxSAT. The types of repairs that are not present did not find a feasible solution. (O) represent an optimal solution.

4.3 HeatShock case study

Considering the complete data set of the HeatShock experimental profile, one can see that the result is similar to the one referring to the Stat vs Exp data set (Figure 4.7). The repair *n* is the only one to find an optimal solution (when considering the ASP implementation), and it requires a lower number of repairs. As said above, for this network when considering the Stat vs Exp data set, the repair *n* has the shortest number of possible repair operations that can be performed and so it is faster to compute

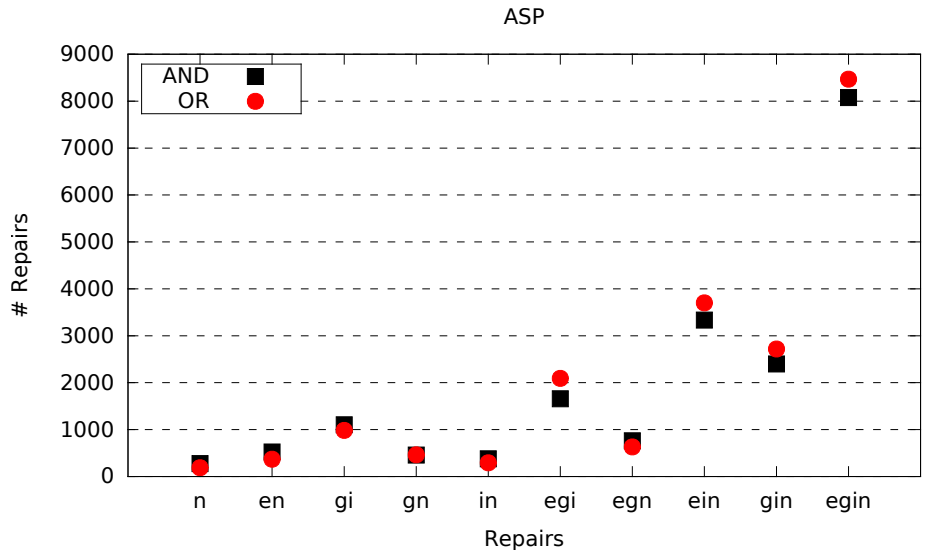


Figure 4.7: Number of repairs necessary to correct the HeatShock data set, using ASP. Repair *n* is the only one to find an optimal solution.

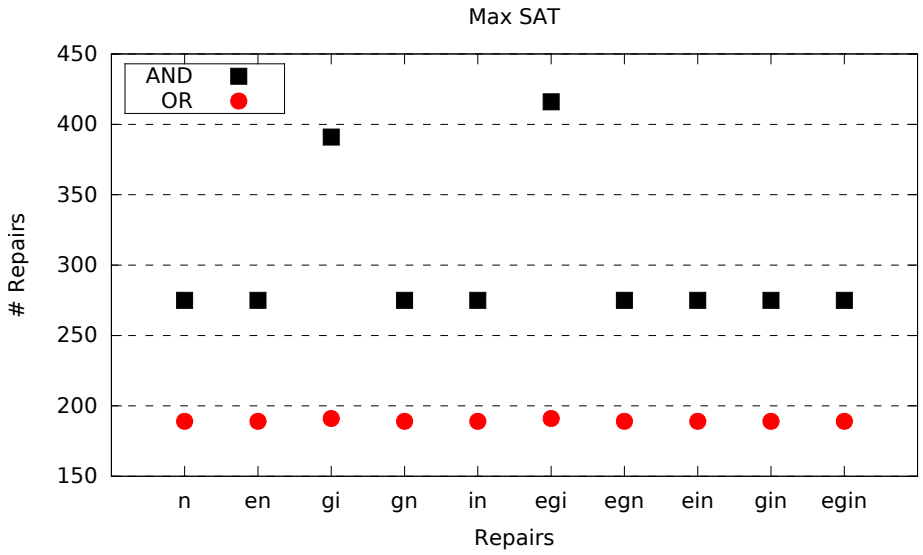


Figure 4.8: Number of repairs necessary to correct the HeatShock data set, using MaxSAT. In this case only the repairs *gi* and *egi* are not optimal for the default function AND. The repair operations that are not present did not find a feasible solution.

a solution if one exists. The main reason for this repair to be successful, in this data set, is that there are many nodes with the value false when all their regulators are all true. It could be possible that with a larger time limit, a different combination would improve the results. But as it can be seen in Figure 4.8 no repair can find a better solution. Figure 4.7 shows the results for the repairs that find a feasible solution. In this case, the difference between the AND function and the OR function as default functions seems marginal. However, it is possible to see in Figure 4.8 that the OR function as default requires less repairs. The repair *n* is the repair that finds a minimal number of repairs. Combining it with other repair operations does not improve the result. Similar to what happens in *Stat vs Exp* data set, the repair *n* is the one requiring the smallest number of repairs. The closest repair that does not contain repair *n*, repair *gi*, requires a bit more repairs.

Figure 4.9 shows the evolution of the time needed to find an optimal solution using repair *n* with different percentages of data for the ASP approach. The time it took to find the optimal solution, when running with the OR function as the default function, is similar in both data sets but for HeatShock it is slightly higher when using the full data set. The time needed to find an optimal solution increases with the percentage of data considered in the data set.

Repair *g* is not able to find any solutions when considering the full data set. However, when considering only a fraction, it finds solutions. Figure 4.11 shows the percentage of satisfiable tests for the different percentage of data using repair *g* and considering the AND function as the default function. One can see that the HeatShock data set has a higher rate of successful repairs but both data sets reduce the percentage of successful repairs tests when the percentage of data increases.

It is expected that repair *g* is able to find some solutions because, as it was said before, the data set has nodes that are repaired with the NAND function, and part of them are satisfiable with an OR function; that is why using the OR function as default requires less repairs.

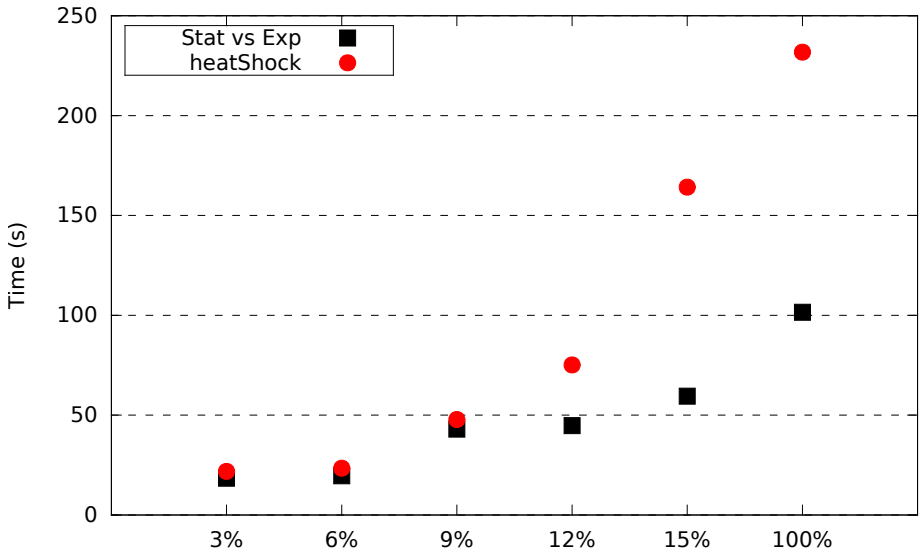


Figure 4.9: Time (in seconds) required to find an optimal solution for repair *n* when running with the OR function as default for the different percentages using the ASP.

Figure 4.10 shows the time required to find a solution to the complete HeatShock data set for both default functions (AND and OR) using the MaxSAT approach. Repair *n* is clearly the one that requires less time. The addition of repairs such as *i* and *e* are the main reason for the longest times. This can be explained since the encoding for both repairs requires the addition of many new clauses. Comparing the execution time necessary to find an optimal solution with the repair *n* with the OR function as the default function using the MaxSAT approach (red dot in the repair *n* column in Figure 4.10) with the one using the ASP approach (the red dot in the 100 % column in Figure 4.9) one can see that the MaxSAT solution is clearly faster. The MaxSAT solution finds an optimal solution in less then 1 second instead of the more than 200 seconds spent by the ASP solution.

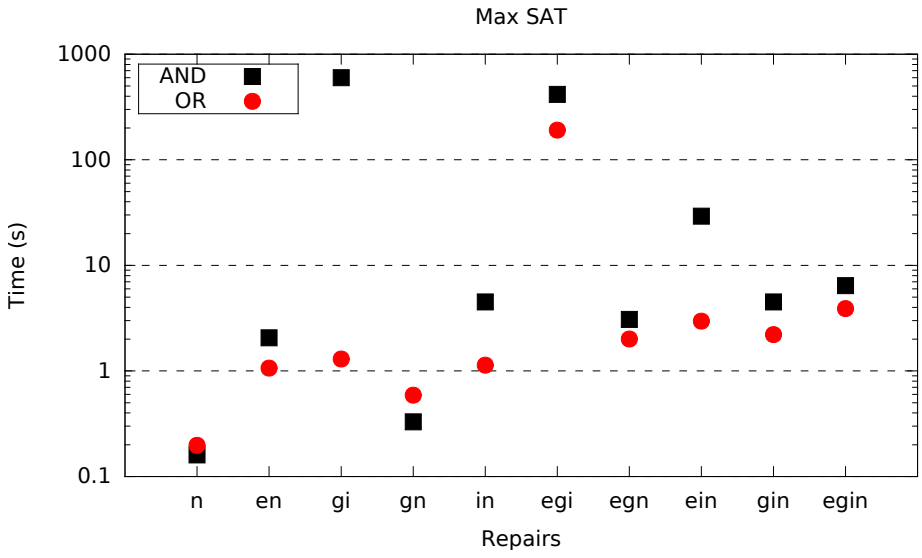


Figure 4.10: Time (in seconds) required to find a solution for the complete HeatShock data using the MaxSAT implementation.

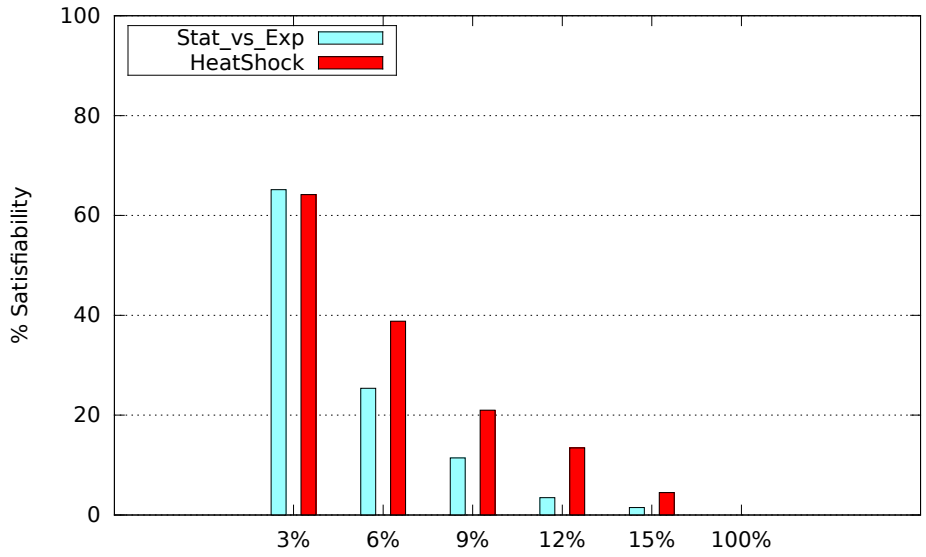


Figure 4.11: Percentage of satisfiable tests for the different percentage of data, using repair *g*, when considering the AND function as default function.

When considering both data sets (combining HeatShock and Stat vs Exp) there are no repair operations that find an optimal solution within the time limit if one is using the ASP solution. As previously discussed using the repair *egi*, one can obtain exactly the same function as the repair *egin* but it may require more repairs to do it. The repair *ein* covers less functions but still achieves a solution within the time limit. When considering the OR function as default function, it is the one that requires the least amount of repairs. The reason behind this is that negating a function, in some case, is cheaper than the same transformation via negating inputs. Table 4.2 shows the number of repairs required for repairing the model. The repair *egin* will at least require the same number of repairs as the repair *egi* as it also includes theses repairs. Even thought these two repairs have the same function coverage (section 3.2), the repair *egin* has more possible repairs to check since it has different ways to obtain the same function. Once again one can see that the model with the OR function as the default function requires less repairs. The last line of the table shows that the *egin* repair is as good as the *ein* repair and better than the *egi* repair. The difference between the minimal number of repairs required to correct both data sets using the repairs *egin* and *egi* is explained by the fact that the repair *egi* may require more repair operations to achieve the same function.

		egi	ein	egin
ASP	AND	925	3225	4360
MaxSAT		326 (O)	471	318 (O)
ASP	OR	809	3302	4747
MaxSAT		238 (O)	237 (O)	237 (O)

Table 4.2: Number of repairs necessary to correct the model when considering both data sets (Stat vs Exp and HeatShock) for both implementations. Only the combination of repairs that find a feasible solution are shown. (O) represent an optimal solution.

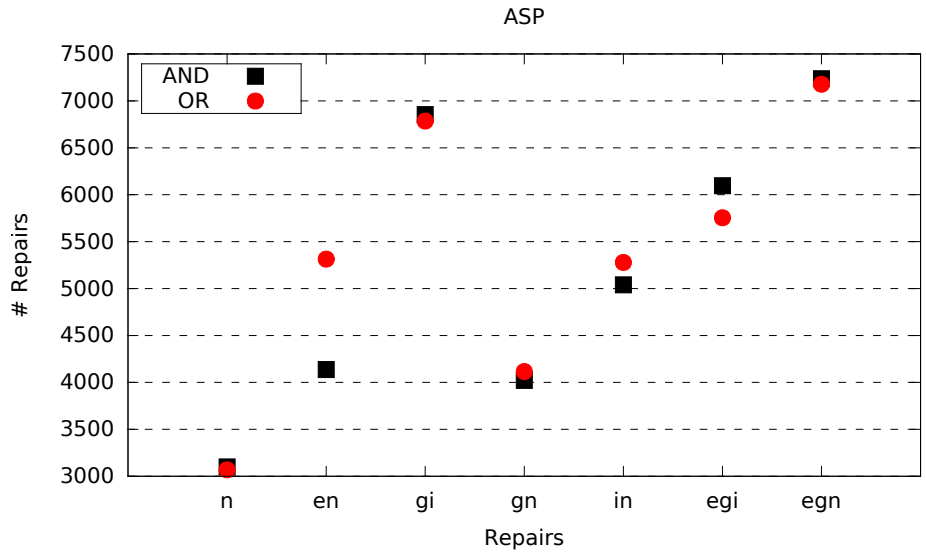


Figure 4.12: Number of repairs, on average, necessary to correct the *Candida albicans* model using the ASP implementation. The types of repairs that are not present did not find a feasible solution or exceeded the memory limit and so no data is available. Repair *n* is the only one to find an optimal solution.

A final remark considering the SCM approach [7] for the HeatShock data set. The combination of the three model repairs *aeg* (adding edges, flipping signs and making node inputs) finds a non optimal solution within the 600 seconds time limit. The SCM was executed using a program implemented in ASP available at http://www.cs.uni-potsdam.de/bioasp/KR10/. Here, the proposed methodology is able to solve all the data sets without compromising much the performance. Most of the execution times of the ASP approach are comparable to the execution times from the most difficult combination of repairs in the SCM implementation.

4.4 *Candida albicans* case study

In order to test a model of a different organism, we considered the *Candida albicans* fungus. Here, since no experimental profile is available, the program tries to generate data which is consistent with the model. If it is impossible to generate data consistent with the model, then the model needs to be revised. It is natural that initially the model does not require any repairs since the model was constructed based on consistent data. In order to evaluate the repairs and see which is the one that better corrects this network, some experimental profiles were randomly generated considering changes in 50% of the data set. These profiles were constructed based on the data from the output of the program when run without any initial profile. The tests were run in the same conditions as for the previous data sets.

Figure 4.12 shows the average number of repairs for the fifteen randomly generated data sets when having either the AND function or the OR function as the default function. Only the repair *n* achieves an optimal solution. The repairs *e, g, eg* do not find any feasible solutions. The combinations of repairs *ein, gin, egin* exceeded the memory limit.

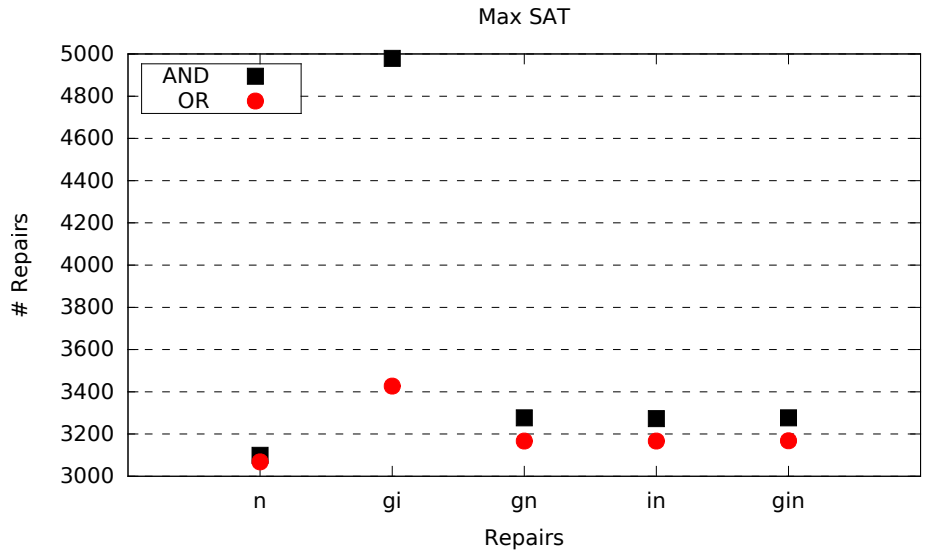


Figure 4.13: Number of repairs, on average, necessary to correct the *Candida albicans* model using the MaxSAT implementation. The repairs contaning *e* were not evaluated. The repair *n* is the only repair that always finds an optimal solution.

Figure 4.13 shows the average number of repairs for the generated data set. The results of the MaxSAT approach show that repairing the model with the OR function as the default function requires less repairs then using the same model but with the AND function as the default function. Note that all repairs containing the repair *e* were not tested since the number of edges involved a very large encoding. The encoding was initially generated with a java program that converted the model and the data into the CNF encoding for a given repair. When running for the *Candida albicans* with a repair containing the repair *e*, the program exceeded the memory heap. This happens due to the amount of calls required to compute all clauses (equation 4.1 shows the number of clauses required for a function with n arguments). This problem was solved by reducing the number of arguments involved in the function calls of the java program, this made the program to no longer exceed the memory heap.

However, the proposed solution did not solve the main problem since the encoding became too big to handle. The main reason for this to happen with this network and not with the one of *Escherichia coli* is that this network has many more functions with a large number of arguments. For example, there is a function with 50 arguments that would require around $1.1 * 10^{15}$ clauses. The ASP solver is able to solve some of the instances containing repair *e* since it does not need to instantiate all possible repairs upfront. The modern solvers try to avoid the instantiation of rules (see smart grounding in section 2.3). In this case the possible repairs, that will never be reached. Even with the smart grounding and the CDCL based solving algorithms, the ASP solution sometimes exceeds the memory limit during the solving process.

Chapter 5

Conclusions

Reasoning over biological networks is a hard problem. One important aspect is to be able to check the satisfiability and to repair a given model in case of inconsistency. There are already several different ways proposed to repair a network. The work shown in this document proposes a new set of repair operations focusing on repairing a Boolean model where the interactions between genes are described by Boolean functions. The repair operations were encoded using both ASP and MaxSAT and both were able to repair networks with any number of arguments. The proposed repairs were able to solve the real life biological networks of *Escherichia coli* and *Candida albicans*. Even when one is considering the data sets of two different experiments of *Escherichia coli* the combination of repair operations can find a suitable answer.

The MaxSAT approach is clearly faster and is able to find a larger number of optimal answers then the ASP implementation. The ASP implementation is easier to change if one wants to add new functionalities. Considering both approaches provided us with the advantage of comparing both implementations, and use them to find coherent answers for the same instances.

Negating a Boolean function (repair *n*) is clearly the repair operation that requires less changes to the original model in order to make it satisfiable. This repair is particularly efficient since there are many nodes with the value false when their regulators are all true and also there are nodes with the value $\mathrm{true/false}$ when the corresponding function is AND/OR and their regulators have different values. Removing regulators of a function blindly, *i.e.* without any constraints, is difficult since there is a blow out of the number of required clauses.

5.1 Achievements

The work presented in this document proposes a set of repair operations to repair an inconsistent biological network described using the Boolean formalism. The repair operations are based on the changing a Boolean function, with the goal of choosing the correct function to fit all data. Even thought the proposed repair operations do not cover all possible functions they were able to solve all networks tested.

The repair operations were encoded using both ASP and MaxSAT. ASP is slower but easier to change.

On the other hand, MaxSAT has a more complex encoding but it is faster to obtain an optimal answer. Both implementations were tested using real life biological networks of *Escherichia coli* and *Candida albicans*. The repair operations proposed were enough to repair all networks. Both implementations allow the use of multiple sets of data when checking and repairing the satisfiability of a network.
The repair operation that allows the removal of regulators is difficult to solve when one is considering a large network like the one from *Candida albicans*.

5.2 Future Work

As future work it is possible to filter the type of functions allowed in the input, *e.g.* making it necessary for the input to be written in a normal form. This constraint does not need to affect the end users since it is possible to use a preprocessing tool to convert the input from the user into a specific normal form. Using a normal form makes it easier to evaluate the functions covered by each repair operation. In the two tools developed in this work the function covered by a repair depends on the starting function (as discussed in section 3.2 for the XOR function).
Even with the positive results from the set of repair operations presented in this document, it is possible to add new repair operations in order to achieve a total function coverage. However, the addition of the functions true and false may complicate the results since it will always find a solution, ignoring all the relations in the model. In the worst case this repair may remove all regulators making all nodes independent which most likely does not make biological sense. Additionally, some other types of repairs may be interesting to study, for example allowing functions with three arguments to become a combination of two different basic functions (AND, OR).
As future work, the choice between different possible repairs could be explored in order to give priority to biologically relevant repairs. As discussed in this document, the encoding for these repairs may enlarge the search space, making it slower. So, another possible direction one can take is to try to reduce the search space by adding biological rules. These rules will try to better characterize the repairs allowed, therefore filtering the space of possible repairs. The blindly removal of regulators from the model is complicated since it enlarges too much the encoding (as discussed in section 4.4). To mitigate this problem one may not allow the blindly removal of regulators, filtering the possibilities based on biological rules, this way avoiding having to enumerate all combinations of possible arguments for a function. (Equation 4.1 shows the number of clauses required to encode all possible functions depending on the number of arguments of the original function.)
Another different possible direction is to try to adapt the algorithms from model based diagnosis [50] to be able to apply the repair operations described in this document. The algorithms developed in this field, originally for hardware circuits validation [51], can be applied to a wide range of problems: software fault [52] localization, debugging of spreadsheets [53] and even analysis of biological systems [54, 55]. *Scrypto* [56][1] is a tool that finds all cardinality minimal diagnosis[2] for a previously described system.

[1] *Scrypto* is available in http://amit.metodi.me/research/mbdsolver/
[2] A minimal diagnosis is the minimal number of components that need to be considered abnormal to make the system description coherent (model) with observations.

However, *Scrypto* only supports partial assignment of the variables describing the system if all the input and output variables are assigned. The tool allows the use of multiple sets of observations but the system will evaluate them separately.

Finally, the set of default functions to be considered could also have a biological support, *e.g.* considering a disjunction over the activators in conjunction with the conjunction of the negated inhibitors, *i.e.* a target is active if there is at least one activator and none of its inhibitors.

Bibliography

[1] Thomas, R., Thieffry, D., Kaufman, M.: Dynamical behaviour of biological regulatory networks: I. Biological role of feedback loops and practical use of the concept of the loop-characteristic state. Bull. Math. Biol. 57(2), 247–276 (1995)

[2] Chaouiya, C.: Petri net modelling of biological networks. Brief. Bioinform. 8(4), 210–219 (2007)

[3] Glass, L., Kauffman, S.: The logical analysis of continuous, non-linear biochemical control networks. Journal of Theoretical Biology 39(1), 103–129 (1973)

[4] Siegel, A., Radulescu, O., Le Borgne, M., Veber, P., Ouy, J., Lagarrigue, S.: Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks. Biosystems 84(2), 153–174 (2006)

[5] Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, chap. 4, pp. 131–153 (2009)

[6] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer set solving in practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)

[7] Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13 (2010)

[8] Kittas, A., Barozet, A., Sereshti, J., Grabe, N., Tsoka, S.: Cytoasp: a cytoscape app for qualitative consistency reasoning, prediction and repair in biological networks. BMC Systems Biology 9, 34 (2015)

[9] Guerra, J., Lynce, I.: Reasoning over biological networks using maximum satisfiability. In: Principles and Practice of Constraint Programming. pp. 941–956. Springer Berlin Heidelberg (2012)

[10] Lemos, A., Monteiro, P.T., Lynce, I.: Repairing boolean regulatory networks using answer set programming. INForum (2016)

[11] Lemos, A., Monteiro, P.T., Lynce, I.: Repairing boolean regulatory networks using answer set programming. Tech. Rep. 5, INESC-ID (2016)

[12] de Jong, H.: Modeling and simulation of genetic regulatory systems: A literature review. Journal of Computational Biology 9(1), 67–103 (2002)

[13] Petri net modelling of biological regulatory networks. Journal of Discrete Algorithms 6(2), 165 – 177 (2008)

[14] Ropers, D., de Jong, H., Page, M., Schneider, D., Geiselmann, J.: Qualitative simulation of the carbon starvation response in escherichia coli. Biosystems 84(2), 124–152 (2006)

[15] Batt, G., Besson, B., Ciron, P.E., de Jong, H., Dumas, E., Geiselmann, J., Monte, R., Monteiro, P.T., Page, M., Rechenmann, F., Ropers, D.: Genetic network analyzer: A tool for the qualitative modeling and simulation of bacterial regulatory networks. In: Bacterial Molecular Networks, pp. 439–462. Springer (2011)

[16] Didier, G., Remy, E., Chaouiya, C.: Mapping multivalued onto Boolean dynamics. Journal of Theoretical Biology 270(1), 177–184 (2010)

[17] Martins, R., Manquinho, V.M., Lynce, I.: Open-wbo: A modular maxsat solver,. In: Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. pp. 438–445 (2014)

[18] Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187-188, 52–89 (2012)

[19] Krentel, M.W.: The complexity of optimization problems. In: Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA. pp. 69–76 (1986)

[20] Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD, Austin, Texas, USA, November 11-14, Proceedings. pp. 13–19 (2007)

[21] Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning, pp. 466–483. Symbolic Computation, Springer (1983)

[22] Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (1965)

[23] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)

[24] Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT 7(2-3), 59–6 (2010)

[25] Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: Qmaxsat: A partial max-sat solver. JSAT 8(1/2), 95–100 (2012)

[26] Lifschitz, V.: What is answer set programming? In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI, Chicago, Illinois, USA, July 13-17. pp. 1594–1597 (2008)

[27] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes). pp. 1070–1080 (1988)

[28] Schaub, T.: Answer set programming. http://www.cs.uni-potsdam.de/~torsten/Potassco/Slides/asp.pdf, accessed: 2015-11-14

[29] Chang, C.L., Lee, R.C.T.: Symbolic logic and mechanical theorem proving. Academic press (2014)

[30] Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR, Vancouver, Canada, May 16-19. Proceedings. pp. 345–351 (2011)

[31] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)

[32] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12. p. 386 (2007)

[33] Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)

[34] Wang, D., Yan, K., Sisu, C., Cheng, C., Rozowsky, J.S., Meyerson, W., Gerstein, M.B.: Loregic: A method to characterize the cooperative logic of regulatory factors. PLoS Computational Biology 11(4) (2015)

[35] Saadatpour, A., Albert, R.: Boolean modeling of biological regulatory networks: A methodology tutorial. Methods 62(1), 3–12 (2013)

[36] Merhej, E., Schockaert, S., Cock, M.D.: Using rules of thumb for repairing inconsistent answer set programs. In: Scalable Uncertainty Management - 9th International Conference, SUM 2015, Québec City, QC, Canada, September 16-18, 2015. Proceedings. pp. 368–381 (2015)

[37] Mobilia, N., Rocca, A., Chorlton, S., Fanchon, E., Trilling, L.: Logical modeling and analysis of regulatory genetic networks in a nonmonotonic framework. In: IWBBIO. LNCS, vol. 9043, pp. 599–612 (2015)

[38] Corblin, F., Tripodi, S., Fanchon, E., Ropers, D., Trilling, L.: A declarative constraint-based method for analyzing discrete genetic regulatory networks. Biosystems 98(2), 91–104 (2009)

[39] Corblin, F., Fanchon, E., Trilling, L.: Applications of a formal approach to decipher discrete genetic networks. BMC Bioinformatics 11, 385 (2010)

[40] Gebser, M., König, A., Schaub, T., Thiele, S., Veber, P.: The bioasp library: ASP solutions for systems biology. In: 22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1. pp. 383–389 (2010)

[41] Melas, I.N., Samaga, R., Alexopoulos, L.G., Klamt, S.: Detecting and removing inconsistencies between experimental data and signaling network topologies using integer linear programming on interaction graphs. PLoS Computational Biology 9(9) (2013)

[42] Thiele, S., Cerone, L., Saez-Rodriguez, J., Siegel, A., Guziolowski, C., Klamt, S.: Extended notions of sign consistency to relate experimental data to signaling and regulatory network topologies. BMC Bioinformatics 16, 345 (2015)

[43] Bay, S.D., Shrager, J., Pohorille, A., Langley, P.: Revising regulatory networks: from expression data to linear causal models. Journal of Biomedical Informatics 35(5-6), 289–297 (2002)

[44] Link, H., Christodoulou, D., Sauer, U.: Advancing metabolic models with kinetic information. Current Opinion in Biotechnology 29, 8–14 (2014)

[45] Comtet, L.: Advanced combinatorics: The art of finite and infinite expansions. p. 187. Springer, Holland (1974)

[46] Gama-Castro, S., Jiménez-Jacinto, V., Peralta-Gil, M., Santos-Zavaleta, A., Peñaloza-Spinola, M.I., Contreras-Moreira, B., Segura-Salazar, J., Muñiz-Rascado, L., Martínez-Flores, I., Salgado, H., et al.: Regulondb (version 6.0): gene regulation model of escherichia coli k-12 beyond transcription, active (experimental) annotated promoters and textpresso navigation. Nucleic acids research 36(suppl 1), D120–D124 (2008)

[47] Allen, T.E., Herrgård, M.J., Liu, M., Qiu, Y., Glasner, J.D., Blattner, F.R., Palsson, B.Ø.: Genome-scale analysis of the uses of the escherichia coli genome: model-driven analysis of heterogeneous data sets. Journal of bacteriology 185(21), 6392–6399 (2003)

[48] Bradley, E.H., Curry, L.A., Devers, K.J.: Qualitative data analysis for health services research: Developing taxonomy, themes, and theory. Health Serv Res 42(4), 1758–1772 (2007)

[49] Roussel, O.: Controlling a solver execution with the runsolver tool. JSAT 7(4), 139–144 (2011)

[50] Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)

[51] Friedrich, G., Stumptner, M., Wotawa, F.: Model-based diagnosis of hardware designs. Artif. Intell. 111(1-2), 3–39 (1999)

[52] Staber, S., Jobstmann, B., Bloem, R.: Diagnosis is repair. Unpublished manuscript (2005)

[53] Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Autom. Softw. Eng. 23(1), 105–144 (2016)

[54] Lucas, P.J.F.: Model-based diagnosis in medicine. Artificial Intelligence in Medicine 10(3), 201–208 (1997)

[55] Bazzan, A.L.: Distributed diagnosis of faults in circuits and biological systems. In: Proceedings of 3rd Workshop on Model-Based Systems (ECAI). pp. 16–20 (2006)

[56] Metodi, A., Stern, R., Kalech, M., Codish, M.: Compiling model-based diagnosis to boolean satisfaction. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada. (2012)

[57] Le Berre, D.: Introduction to SAT: History, algorithms, practical considerations. SAT/SMT Summer School Semmering, Austria (2014)

[58] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)

[59] Marques-Silva, J., Janota, M.: CDCL SAT Solvers & SAT-Based problem solving. SAT/SMT Summer School 2013 Aalto University, Espoo, Finland (2013)

[60] Marques-Silva, J., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD. pp. 220–227 (1996)

[61] da Terra Neves, M.Â.: A Distributed MaxSAT Solver. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (2014)

[62] Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Theory and Applications of Satisfiability Testing - SAT, 9th International Conference, Seattle, WA, USA, August 12-15, Proceedings. pp. 252–265 (2006)

Appendix A

SAT Algorithms

Davis-Putnam algorithm

This algorithm was originally proposed by Davis and Putnam [23] in 1960 and is based on the resolution rule. For every clause having a variable x and every clause containing the negation of that variable, $\neg x$, the resolution rule applies. All the clauses that have the variable x are removed, and the resolvents are added to the formula. At this step if the result is an empty clause then the new formula is unsatisfiable, otherwise it is satisfiable.
This algorithm was not able to solve many real life SAT instances, so the authors proposed an improved version without using the resolution rule. To explain the new algorithm is necessary to define a few more concepts.

Definition 19. Unit propagation is a procedure that iteratively satisfies the unit clauses.

Definition 20. A literal is considered pure if this literal only appears as a positive or negative literal in the formula.

For example the formula $\mathrm{f1}$,

$$\mathrm{f1} = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge x_3$$

has two pure literals, x_1 and x_3.

Davis-Putnam-Logemann-Loveland Algorithm

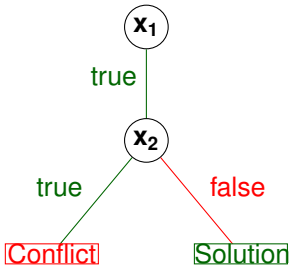
The pseudo-code for this algorithm [58] is shown in Algorithm 1. The DPLL algorithm starts by checking the presence of unit clauses and applying unit propagation. In the original version of this algorithm, the clauses containing pure literals are removed since their existence does not change the satisfiability of the formula (removing the pure literals corresponds to assigning the literal to true). The algorithm in the line 6, will assign a value to an unassigned variable and in the next line the algorithm will recursively call the same function returning to the line 2. If a unit clause is found, it will assign that literal with the necessary value to make the clause satisfied and execute the unit propagation procedure. If a conflict (an assignment that causes a formula to be unsatisfied) arises in the process and if it is possible to

Algorithm 1 DPLL [57]

1: DPLL (formula)
2: Apply unit propagation
3: **if** conflict identified **then return** unsatisfiable
4: [Optional] Remove clauses containing pure literals
5: **if** No more unassigned variables **then return** satisfiable
6: $x \leftarrow Select$ unassigned variable
7: **if** DPLL ($formula \vee x$) = satisfiable **then return** satisfiable
8: **else return** DPLL ($formula \vee \neg x$)

Decision Level	Variable	Propagation
1	x_1	
2	$\neg x_2$	$\rightarrow \neg x_3$

(a)



(b)

Figure A.1: On the left a decision table of the last iteration of the DPLL algorithm. On the right the search tree from the algorithm.

undo the last decision the algorithm will backtrack and flip the variable value. If a conflict still exists after flipping the variable value, it will chronologically revert the previous decision. If it is impossible to change an assignment the algorithm will end because the formula is unsatisfiable. When there are no more unassigned variables the formula is satisfied [59].

Considering the formula $\mathrm{f3}$, the DPLL algorithm (without line 4) will, first, assign the variable x_1 with the value true. As there are no unit clauses, no conflicts and there are still unassigned variables the algorithm will assign a value to another variable. When assigning the variable x_2 as true it will form a unit clause x_3. So x_3 will be assigned as true but this generates a conflict. It is possible to undo the last round of assignments, backtrack and flip the value of x_2. Now x_2 has been assigned the value false (line 8) and again a unit clause is found. x_3 will be assigned as false (line 2) and the conflict removed. As there are no more unassigned variables the algorithm ends and concludes that the formula $\mathrm{f2}$ is satisfied. Figure A.1 on the right has the final search tree, and on the left a table showing the decision level for the last iteration of the algorithm. Decision levels with more than one variable mean that the other variables present were assigned by unit propagation, and thus are not present in the tree.

$$\mathrm{f2} = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$$

CDCL [5, 60, 59] is an algorithm designed to solve SAT that analyses conflicts in order to reduce the search space. In order to explain the CDCL algorithm it is important to define what is an implication graph.

Definition 21. An implication graph is a direct acyclic graph composed by a set of nodes N and a set of edges E, where each node in N represents a variable assignment and if the assignment of a variable generates a unit clause the graph will have an edge, representing the unit clause, between the nodes that caused the unit clause and the node that represent the variable from the unit clause.

Conflict-Driven Clause Learning

CDCL is an algorithm to solve SAT instances, which implements the DPLL algorithm and when a conflict is discovered it will learn from it. CDCL, during its execution, creates an implication graph that it will be used to "find" the new clause. Algorithm 2 shows the structure of a CDCL solver. The unit propagation used in lines 2 and 8 of the algorithm is exactly the same as in the DPLL algorithm. The backtracking is not chronological like the one in DPLL because, based on the conflict analyses, it will compute the place in the search tree (decision level) to perform a backjump (the jump happens in line 12). The ConflictAnalysis is a procedure that analysis the conflict. This procedure starts on the conflict clause and it will transverse the graph "visiting" all the assigned variables that are relevant for this conflict and identifying their antecedents (the unit clause used for implying the variables). In the basic version of this algorithm this procedure will continue until the most recent decision variables are reached and then it will learn a new clause based on the decision variables that originated this conflict. Many strategies can be used to optimize this algorithm and in particular to improve the termination condition, for example using Unit Implication Points (UIPs) [5]. The simplest version of this procedure can be understood as a sequence of multiple resolution operations involving the clauses that have variables that were "visited". In this case, all the operations would generate a temporary new clause. When there are no more possible operations the clause would be added. At the end of this procedure, if it is not possible to perform a backjump, the formula is unsatisfiable (line 11). AllVariablesAssigned is a test in order to know if the formula has yet an unassigned variable. If all the variables have been assigned it means that the formula is satisfied.

Again starting with formula $\mathrm{f2}$, all the steps will be the same as in the application of DPLL until the detection of the conflict. This conflict was caused by the clauses $\neg x_2 \vee x_3$ and the $\neg x_2 \vee \neg x_3$. After traversing the generated graph the algorithm will learn a new clause based on the decision variable of this level (see figure A.1 on the left), $\neg x_2$. Obviously the result will be the same, and in this example there will be no difference in the backtrack phase because the computed jump will be the same as the chronological jump (to decision level 1).

It is also important in these types of algorithms to check for UIPs in order to reduce the size of the new learned clause. UIP represents a different assignment that at this step will produce the same conflict. It is possible to use the implication graph to identify UIPs and it is possible to stop the transverse action when an UIP is found. In figure A.2 we can see that the node representing the positive literal x_7 is an UIP. Using the UIP we can learn a smaller clause $\neg x_7 \vee x_8$ instead of learning $x_1 \vee x_3 \vee x_8$.

$$\mathrm{f3} = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$$

Algorithm 2 CDCL [5]

```
1: CDCL(formula)
2: Apply unit propagation
3: if conflict identified then return unsatisfiable
4: decisionLevel ← 0
5: while Not AllVariablesAssigned(formula) do
6:     x ← Select unassigned variable
7:     decisionLevel ← decisionLevel + 1
8:     Apply unit propagation
9:     if conflict identified then
10:        c ← ConflictAnalysis( formula )
11:        if c < 0 then return unsatisfiable
12:        else Backtrack( c )
13:            decisionLevel ← c
14: return satisfiable
```

$$= (x_1 \vee x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee x_4 \vee x_7) \wedge (\neg x_7 \vee \neg x_5) \wedge (x_8 \vee \neg x_6 \vee \neg x_7) \wedge (x_5 \vee x_6)$$

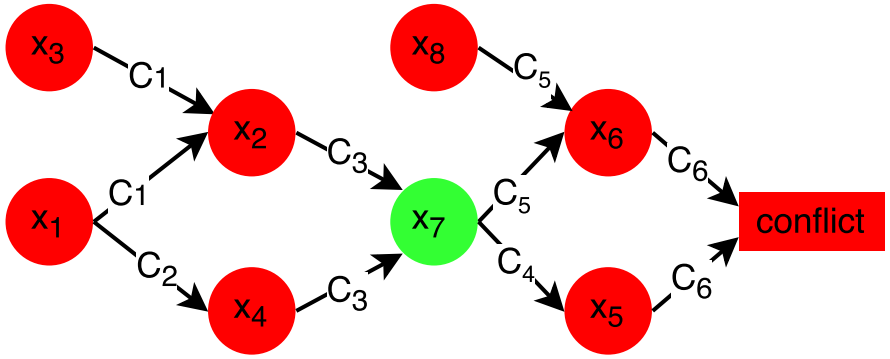


Figure A.2: An implication graph showing a conflict when assigning the formula $\mathrm{f3}$ [5].

Appendix B

MaxSAT Algorithms

Linear Search Sat-Unsat Algorithm

A different new literal, $\mathrm{r_i}$, is added to each clause of a formula. These new literals are used to relax the formula. The linear Search Sat-Unsat algorithm uses a SAT-solver as a black box (SAT oracle) and adds a new constraint on the upper bound that can be written as equation (B.1), where c is the number of clauses and w_i is the weight of the clause i. In the case of applying this algorithm to a non weighted MaxSAT instance we can consider that w_i has the same value for every clause i. This algorithm starts by assigning the upper bound as the sum of the weight of all the clauses. The algorithm ends when the SAT solver returns that the formula is unsatisfiable; until then the algorithm iteratively calls the solver with decreasing upper bounds. At each iteration the upper bound value is updated by summing the cost of all the clauses that have the new literals assigned as true to check if it is possible to reduce the cost of the unsatisfied clauses [61].

$$\sum_{i=1}^{c} w_i r_i < upperBound \tag{B.1}$$

Considering the formula $\mathrm{f4}$,

$$\mathrm{f4} = x_1 \wedge \neg x_1 \wedge (\neg x_1 \vee x_2) \wedge \neg x_2$$

the algorithm will add a different new literal to each clause. From now on the algorithm will work with $\mathrm{f5}$.

$$\mathrm{f5} = (x_1 \vee r_1) \wedge (\neg x_1 \vee r_2) \wedge (\neg x_1 \vee x_2 \vee r_3) \wedge (\neg x_2 \vee r_4)$$

For this example we are going to consider that all the clauses have the same weight, and so the upper bound is initialized as the number of clauses plus one (five). The SAT solver will return a satisfied formula and only $\mathrm{r_1}$ and $\mathrm{r_4}$ are assigned as true. So the upper bound will be updated to two. In the next call the SAT solver will return that the formula is unsatisfiable since it is impossible to satisfy the formula without removing two clauses and breaking the constraint imposed by the equation (B.1).

Core-guided MaxSAT

The objective of this algorithm is to remove only the relevant clauses, i.e only relax the clauses from unsatisfiable cores. An unsatisfiable core is a subset of the formula that is still unsatisfiable. This algorithm uses a SAT solver but requires that this solver returns the unsatisfiable core. The pseudo-code for the algorithm proposed by Fu and Malik [62], a core-guided approach to solve partial MaxSAT, is shown in algorithm 3.

Algorithm 3 Partial MaxSAT - Core Guided Unsat-Sat algorithm [62]

```
1:  status ← unsatisfiable
2:  while status = unsatisfiable do
3:      (core, status) ← SAT solver (formula)
4:      if status = unsatisfiable then
5:          S ← ∅
6:          for each Clauses c_i ∈ core do
7:              if c_i is a soft clause then
8:                  r_i ← Allocate a new variable
9:                  c ← c ∨ r_i
10:                 S ← S ∨ { r_i }
11:         if S = ∅ then
12:             return unsatisfiable // all clauses in core are hard
13:         else
14:             Add clauses to enforce: Σ_{r∈S} r ≤ 1
15: return assignment that satisfies the formula
```

The algorithm calls the SAT solver to obtain the unsatisfiable core (line 3) and relaxes a soft clause until the formula is satisfiable (line 2). Considering that all the hard clauses have to be satisfied in the partial MaxSAT the algorithm only relaxed the soft clauses present in the core (lines 6 to 10). If an unsatisfiable core exists and no relaxing variables are used, the algorithms return unsatisfiable (line 11). In line 13 the algorithm guarantees that only one variable in the unsatisfiable core is satisfied (one-hot constraint). If the formula is satisfiable the algorithm returns an assignment that satisfies the formula considering the assignment returned by the SAT solver and all relaxed variables.