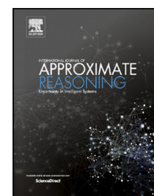
International Journal of Approximate Reasoning 83 (2017) 243–264

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar



Repairing inconsistent answer set programs using rules of thumb: A gene regulatory networks case study ☆

Elie Merhej [a,*], Steven Schockaert [b], Martine De Cock [c,d]

[a] Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Krijgslaan 281, 9000 Ghent, Belgium
[b] School of Computer Science and Informatics, Cardiff University, 5 The Parade, Cardiff CF24 3AA, United Kingdom
[c] Center for Data Science, University of Washington Tacoma, 1900 Commerce Street, Tacoma, WA 98402-3100, United States
[d] Bioinformatics Institute Ghent, Ghent University, Technologiepark 927, 9052 Ghent, Belgium

ARTICLE INFO

Article history:
Received 14 February 2016
Received in revised form 18 December 2016
Accepted 25 January 2017
Available online 30 January 2017

Keywords:
Answer set program
Gene regulatory network
Inconsistency repair
Rule of thumb
Aggregation method

ABSTRACT

Answer set programming is a form of declarative programming that can be used to elegantly model various systems. When the available knowledge about these systems is imperfect, however, the resulting programs can be inconsistent. In such cases, it is of interest to find plausible repairs, i.e. plausible modifications to the original program that ensure the existence of at least one answer set. Although several approaches to this end have already been proposed, most of them merely find a repair which is in some sense minimal. In many applications, however, expert knowledge is available which could allow us to identify better repairs. In particular, we consider the scenario where this expert knowledge is formulated as rules of thumb, but no training data is available to learn how these rules of thumb interact. The main question we address in this paper is whether we can then still aggregate the rules of thumb in a useful way. In addition to standard aggregation techniques, we present a novel statistical approach that assigns weights to these rules of thumb, by sampling, in a particular way, from a pool of possible repairs. In particular, we evaluate how frequently each given rule of thumb is violated in the sample of repairs, and use the Z-score of this distribution to set the weight of that rule. We analyze the potential of using expert knowledge in this way, by focusing on a specific case study: Gene Regulatory Networks. We describe the rules of thumb that express available expert knowledge from the biological literature and explain how they can be encoded while repairing inconsistencies. Finally, we experimentally compare the proposed repair strategies using rules of thumb against the baseline strategy of identifying minimal repairs.
© 2017 Elsevier Inc. All rights reserved.

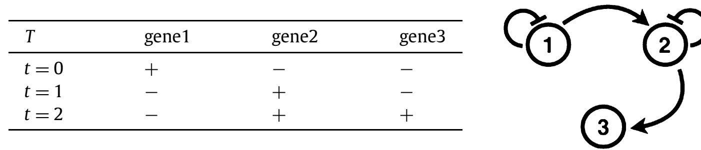
1. Introduction

Answer Set Programming (ASP) is a form of declarative programming mainly oriented towards NP-hard problems [1]. It enables non-monotonic reasoning by virtue of a negation-as-failure operator with a purely declarative semantics [2]. ASP programs describe search or optimisation problems as a set of rules. This set of rules is fed to answer set solvers that find stable models (i.e. answer sets) which correspond to the solutions of the considered problem. Answer set programs

☆ This paper is part of the virtual special issue on the Ninth International Conference on Scalable Uncertainty Management (SUM 2015), edited by Christoph Beierle and Alex Dekhtyar.

* Corresponding author.
E-mail addresses: elie.merhej@ugent.be (E. Merhej), schockaerts1@cardiff.ac.uk (S. Schockaert), mdecock@u.washington.edu (M. De Cock).

http://dx.doi.org/10.1016/j.ijar.2017.01.012
0888-613X/© 2017 Elsevier Inc. All rights reserved.



T	gene1	gene2	gene3
$t=0$	+	−	−
$t=1$	−	+	−
$t=2$	−	+	+

Fig. 1. A time-series table which is inconsistent with a given GRN. Edges with pointed end points denote activations. Edges with flat end points denote inhibitions. The edge $2 \dashv 2$ causes the inconsistency.

are sometimes also used to simulate systems (e.g. for solving planning problems [3,4]), in which case answer sets typically correspond to sequences of states.

We are interested in the case where ASP programs have no answer sets. We call these programs inconsistent, and we look for ways to restore their consistency. For example, in a search problem, having no answer sets could mean that the problem is over-constrained, and we may want to look at ways to relax the problem. In applications where ASP programs simulate a system, inconsistencies could mean that the rules describing the system are not in agreement with available observations, and we may want to find a way to adapt the description of the system. In this paper, we will focus on the latter type of ASP programs.

While different methods exist for repairing ASP programs, most of them are based on finding some sort of minimal repair, e.g. adding or removing the smallest number of facts to ensure that the program has at least one answer set [5–7]. While this is a reasonable principle in the absence of any further information, in real-world applications we often have access to some kind of expert knowledge about the system being modelled that can be exploited to identify the most plausible repair, which may not necessarily be minimal, as we will see in Section 8.

To demonstrate this idea, let us consider the biological setup in Fig. 1. The figure depicts a table containing observed time-series data about which of three genes were active at different time points, and a draft of a Gene Regulatory Network (GRN) which might not be correct. A GRN is a directed graph that represents the way a group of genes affect one another. GRNs can be modelled in different ways [8–10], with one popular model being Boolean networks [11]. Treating a GRN as a Boolean network implies that an edge from gene X to gene Y can either represent a positive regulation, which means that X activates Y, or a negative regulation, which means that X inhibits Y. The data from the table in Fig. 1 is encoded in ASP as facts of the form $gene(X)$ to represent every gene, and $active(X,T)$ or $inactive(X,T)$ to encode whether a gene X is active or inactive at time step T respectively. In particular, we consider the following facts:

$$\begin{array}{lll} gene(1). & gene(2). & gene(3). \\ active(1,0). & inactive(2,0). & inactive(3,0). \\ inactive(1,1). & active(2,1). & inactive(3,1). \\ inactive(1,2). & active(2,2). & active(3,2). \end{array} \tag{1}$$

Similarly, the GRN in Fig. 1 is encoded in ASP as a set of facts of the form $edgeInit(X,Y,1)$ or $edgeInit(X,Y,-1)$ to indicate that there exists an edge between gene X and gene Y with a positive regulation or negative regulation respectively:

$$edgeInit(1,1,-1).\ \ edgeInit(1,2,1).\ \ edgeInit(2,2,-1).\ \ edgeInit(2,3,1). \tag{2}$$

The semantics of Boolean networks are as follows. If gene X is active at a specific time step, and X activates gene Y, then Y becomes active in the next time step. Similarly, if X is active and X inhibits Y, Y becomes inactive in the next time step. These activation rules can be encoded in ASP as follows:

$$\begin{aligned} activates(X,Y) &\leftarrow edgeInit(X,Y,1). \\ inhibits(X,Y) &\leftarrow edgeInit(X,Y,-1). \\ active(Y,T+1) &\leftarrow activates(X,Y),\ active(X,T). \\ inactive(Y,T+1) &\leftarrow inhibits(X,Y),\ active(X,T). \end{aligned} \tag{3}$$

The facts in (1)–(2) and the rules in (3) represent a simplified program that encodes GRNs in ASP. More details about activation rules, as well as the entire ASP encoding of this setup are provided in Section 6. The GRN graph might have missing edges and/or erroneous edges due to the complexity of network generation methods [12,13], and as a result it might be inconsistent with the observed experimental data in the table. The task at hand is to repair the network to make it consistent with the table. Note that since the network and table are encoded as an ASP program, this problem boils down to repairing an inconsistent ASP program. A common method of repair is to find the smallest number of modifications to the graph that makes it consistent with the table. Based on the GRN and table in Fig. 1, since gene 2 stays active from $t=1$ to $t=2$, a possible minimal repair consists in removing the edge $2 \dashv 2$. The repaired network is shown in Fig. 2(a).

However, there is a known property about GRNs which states that the diameter (i.e. the length of the longest of the shortest paths between two nodes in the graph) of a GRN tends to be very small. Taking this information into account, we

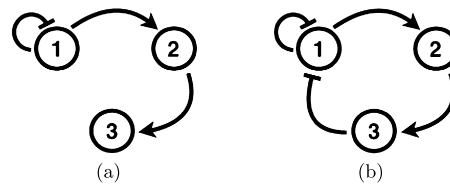


Fig. 2. Two possible repairs for the GRN from Fig. 1. (a) A minimal repair (diameter = 2). (b) A repair that minimizes the diameter of the graph (diameter = 1).

may consider that the repaired network in Fig. 2(b) is actually more plausible. Notice that this repair is not minimal (we removed the edge $2 \dashv 2$ and added the edge $3 \dashv 1$), but the diameter of this new graph (diameter = 1) is smaller than the one for the previous repair (diameter = 2).

The aim of this paper is to assess the viability of using informal, expert-provided rules of thumb for repairing inconsistent ASP programs. While we only consider GRNs in our experiments, the proposed method is entirely generic (see Section 4), being applicable to any setting where expert knowledge can be formalized in ASP [14–16]. We focus in particular on the case where the only available information to guide the repair process comes in the form of informal rules of thumb. While it is clear that learning the weights of these rules from training data would yield better repairs, we are interested in the case where training data is hard to obtain. In this scenario, setting the weights for every rule of thumb becomes a difficult task. To address this problem, we present different techniques for aggregating the evidence that is provided by the number of times each rule of thumb is violated in a given repair, and we show how these techniques can be encoded in ASP. In addition to standard aggregation techniques, such as leximin, leximax, or voting approaches, we present a novel statistical approach that tries to learn the importance of every rule of thumb in an unsupervised way. In particular, this method compares the total penalty of each rule against the expected total penalty, using the Z-score, with the expected penalty being estimated from a set of randomly sampled repairs.

This paper is an extended version of our work published in [17]. We extend our previous work by presenting a general description of the method of repairing inconsistencies using rules of thumb, and show how it can be applied to different domains. We also explore and experimentally compare different approaches of aggregating rules of thumb while repairing an ASP program (Sections 5, 8). The paper is structured as follows. In Section 2, we present an overview of related work. In Section 3, we provide some background on answer set programming. In Section 4, we define rules of thumb, and give a general description of the method of repairing inconsistencies using rules of thumb. In Section 5, we present different approaches for aggregating rules of thumb to identify optimal repairs for an inconsistent ASP program. In Section 6, we introduce the considered application domain. In particular, we show how the compatibility between Gene Regulatory Networks (GRNs) and time-series tables can be checked using an ASP program. Moreover, we explain a method for repairing conflicts, which is based on finding a minimal set of changes to the GRN that makes the resulting answer set program consistent. In Section 7, we describe rules of thumb that express available expert knowledge from the biological literature about GRNs, and we show how they can be encoded in ASP. Subsequently, in Section 8, we discuss our experimental results. Finally, we conclude in Section 9.

2. Related work

Our work is related to many areas of research. ASP has been widely used as a language to detect and repair inconsistencies. This is mainly due to its high expressiveness, declarative nature and flexibility. For example, [16] uses ASP to find semantic inconsistencies (i.e. concepts with erroneous synonymy) in medical language systems. In [18] an application is described where ASP is used for data integration, which deals with inconsistent and incomplete databases. ASP is also used in [6] to retrieve consistent information from inconsistent databases using queried repairs and exceptions.

Several authors have focused on repairing inconsistent answer set programs. The most common method of repair is the minimal repair method used in [19,20,6,5]. The basic idea behind this method is to find a minimal subset of rules whose removal restores consistency to the program. In [21], a technique of dynamic consistency checking for computing answer sets in inconsistent ASP programs is presented. Under this method, only constraints that are deemed relevant to partial answer sets (i.e. subsets of actual answer sets) are tested, allowing inconsistent knowledge bases to be successfully queried. In [22,23], different classifications of “errors” in the ASP program are introduced. Additionally, interactive debugging tools in the form of algorithms-based methods, query-based methods and a tagging technique are proposed. In [24], the idea of belief revision in logic programming under the answer set semantics is proposed: given logic programs P and Q, the goal is to determine a program R that corresponds to the revision of P by Q. This allows solving inconsistencies that may arise when adding new formulas to an existing ASP program. This approach is again based on a form of minimal repair, as the revised program R is the “closest” program to P that is also consistent. In [25], the notion of paracoherent answer sets is presented to repair inconsistent ASP programs. These answer sets are so-called semi-stable models based on a modification of ASP programs, called epistemic transformation.

Biological networks have been previously considered as an application for ASP by several authors. In [26], biological networks are modelled by action languages via ASP. In [27], ASP is used to model GRNs as Boolean networks, similarly

to how we modelled GRNs in (1)–(3). The most closely related research was presented in [28,5,29], where ASP is used to encode GRNs in a Boolean setting, and to detect and repair inconsistencies found in these GRNs. The main difference with our approach is that these existing methods only consider the minimal repair method (which we use as a baseline method in Section 8).

The idea of repairing inconsistencies using soft constraints or "rules of thumb" as we do in this paper, is closely related to the motivation for using Markov Logic in many applications. Markov Logic, presented first in [30], combines first-order logic with probabilistic graphical models. Syntactically, a Markov Logic Network (MLN) is a set of weighted first-order logic formulas. These weights represent the importance of the corresponding formulas. However, the weights are generally difficult to interpret and therefore difficult to set manually. In practice, the weights are almost always learned from training data [31–33]. In [34], Markov Logic is combined with Allen's interval calculus to determine the most consistent subset of an inconsistent database. However, that method needs a database with predefined confidence scores for every fact. In [35], MLNs are used to encode domain knowledge for resolving ambiguities and inconsistencies in extracted information, and for merging multiple information sources. This method relies on weight learning from training data to set the weights of the MLN rules. In areas such as information extraction, the use of Markov logic for repairing inconsistencies is relatively common [36–38]. This use of Markov logic is different from our method as we assume that there is no training data available to learn confidence weights for each soft constraint.

The idea of adding weights to ASP rules has been widely studied. In [39], weights are added to ASP rules to represent the importance of the rules. In [40–43], answer set programming (ASP) and possibility theory is combined to form possibilistic answer set programming (PASP), where weights are used to determine the certainty levels of the ASP rules. In [44], a language is introduced that combines ASP semantics with Markov Logic. The idea of adding soft constraints with varying weights to restore consistency is discussed, with an emphasis on the difficulty of setting these weights in the absence of training data.

Finally, different approaches have been proposed for modeling preferences in answer set programming. The research in [45–47] presents the notion of strong and weak answer sets depending on how strongly encoded preferences are taken into account. In the case of inconsistency, preference rules of the form "rule A has higher priority over rule B" are used to resolve conflicts. This method explicitly models preferences between constraints in order to relax over-constrained optimization or search problems. This is different from our repair approach since it requires explicit preference rules, which we don't require. In [48,49], different applications of preferences using ASP are discussed. In order to represent preferences, an extension of ordered disjunction programs is used. These methods also require certainty scores for the facts and rules of an inconsistent program, which is not needed for our methods.

3. Answer Set Programming

Answer Set Programming (ASP) is a declarative problem solving language [2,1], which requires users to describe a problem as a set of rules. An ASP rule has the form

$$h_1 \mid \ldots \mid h_i \leftarrow a_1, \ldots, a_j, not\ b_{j+1}, \ldots, not\ b_k, \tag{4}$$

where $h_1, \ldots, h_i$, $a_1, \ldots, a_j$ and $b_{j+1}, \ldots, b_k$ are called *atoms*. Let r be an ASP rule of the form (4). We call $head(r) = \{h_1, \ldots, h_i\}$ the *head* of the rule r and $body(r) = \{a_1, \ldots, a_j, not\ b_{j+1}, \ldots, not\ b_k\}$ the *body* of r. Let $body^+(r) = \{a_1, \ldots, a_j\}$ and $body^-(r) = \{b_{j+1}, \ldots, b_k\}$. The "|" in the head of a rule represents a disjunction, while the "," in the body of a rule represents a conjunction. If $body(r) = \emptyset$, then r is called a *fact*. For convenience, the symbol $\leftarrow$ is often omitted when writing facts in ASP. If $head(r) = \emptyset$, then r is called a constraint. Constraints act as filters on the possible answer sets. Answer set programs will often follow a generate-and-test methodology, in which a set of rules is used to generate candidate solutions and constraints are then used to filter these candidates. The keyword *not* represents *negation-as-failure* in ASP, where *not a* intuitively holds whenever we cannot derive that a holds. An answer set program Π is a set of ASP rules of the form (4). A set of atoms X is closed under Π if for any rule $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$. The smallest set of atoms closed under Π is denoted by $Cn(\Pi)$. The reduct Π^X of Π relative to X is defined by

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

A set X of atoms is called an *answer set* (i.e. stable model) of Π if $Cn(\Pi^X) = X$.

Example 1. Let Π be the answer set program formed by the rules "$a \leftarrow not\ b$" and "$b \leftarrow not\ a$". For $X = \{a\}$, $\Pi^X =$ "$a \leftarrow$" and $Cn(\Pi^X) = \{a\}$. Since $Cn(\Pi^X) = X = \{a\}$, $\{a\}$ is an answer set of Π. Similarly, for $X = \{b\}$, $\Pi^X =$ "$b \leftarrow$" and $Cn(\Pi^X) = \{b\}$. Hence $\{b\}$ is also an answer set of Π.

In practice, it is often easier to encode ASP programs using first-order rules such as $R(X_1, X_2, X_3) \leftarrow Q(X_1, X_2),\ not\ S(X_3)$. Such rules should be seen as a compact representation of a set of ASP rules, called the groundings of the first-order rule, which are obtained by considering all possible instantiations of the variables by constants appearing in the program. An ASP solver (e.g. *clasp* [50]) is then used to find the answer sets of the ground program. Typically, ASP programs are specified such that there is a one-to-one mapping between the answer sets of the program and the solutions of the problem being

modelled. In this paper, we are interested in ASP programs that have no answer sets. We call such programs "inconsistent ASP programs".

Example 2. Let Π be the answer set program formed by the rule "$a \leftarrow not\ a$". For $X = \{\emptyset\}$, $\Pi^X =$ "$a \leftarrow$" and $Cn(\Pi^X) = \{a\}$. For $X = \{a\}$, $\Pi^X =$ " " and $Cn(\Pi^X) = \{\emptyset\}$. Hence, there is no set X that satisfies the relation $Cn(\Pi^X) = X$. Therefore, Π has no answer sets. We call Π an inconsistent ASP program.

Most ASP solvers offer extensions in the form of simple operations that can be used in the ASP rules. These extensions include "aggregates" and "optimization statements". Aggregates are special predicates which behave like functions that evaluate to some value. For example, the solver *clasp* supports the aggregates "*#count*", and "*#sum*", to name a few. Optimization statements extend the basic question of whether a set of atoms is an answer set, to whether it is an optimal answer set. For example, the solver *clasp* supports the optimization statements "*#minimize*" and "*#maximize*", which have the effect that only answer sets that minimize or maximize some criteria are considered optimal. Moreover, priorities in the form of integers can be associated with optimization statements, which allows the use of multiple instances of these statements within the same program.

4. Repairing inconsistencies using rules of thumb

In this section, we give a general description of a repair method that consists in using rules of thumb to repair an inconsistent ASP program. Let P be an inconsistent ASP program, F_P the rules r of P such that $body(r) = \emptyset$ (i.e. the facts of P), $R_P = P \setminus F_P$ and Γ_P the set of all the literals in P, defined by $\Gamma_P = \bigcup_{r \in P} head(r) \cup body(r)$. We assume that two disjoint sets of facts F^{inc} and F^{con} are defined a priori, such that for any program P modeling the considered type of problem, it holds that $F_P \subseteq F^{inc} \cup F^{con}$. The set F^{inc} contains the facts that may be causing an inconsistency in a given program P. For example, in a biological application, F^{inc} may contain facts that relate to speculations about an underlying model while F^{con} may contain facts that relate to reliable experimental observations. Repairing P then consists of adding and/or removing facts of F^{inc} from F_P to create an ASP program P_{mod} that is consistent.

Definition 1. Repair

A repair of an inconsistent answer set program P is a pair (X, Y) of sets of facts that satisfies the following conditions:

- $X \cap Y = \emptyset$
- $X, Y \subseteq F^{inc}$
- $P_{mod} = (F_P \setminus Y) \cup X \cup R_P$ is an ASP program that has at least one answer set

In order to find a repair of an inconsistent answer set program, we extend the program with rules that guess the candidate sets X and Y. The answer sets of this extended program will then encode the pairs (X, Y) that correspond to valid repairs. Consider the following example: let $F^{inc} = \{f(1), f(2)\}$. Then, all possible candidate sets are:

- $C_1 : X = \{f(1)\},\ Y = \{f(2)\}$
- $C_2 : X = \{f(1)\},\ Y = \emptyset$
- $C_3 : X = \{f(2)\},\ Y = \{f(1)\}$
- $C_4 : X = \{f(2)\},\ Y = \emptyset$
- $C_5 : X = \{f(1), f(2)\},\ Y = \emptyset$
- $C_6 : X = \emptyset,\ Y = \{f(1), f(2)\}$

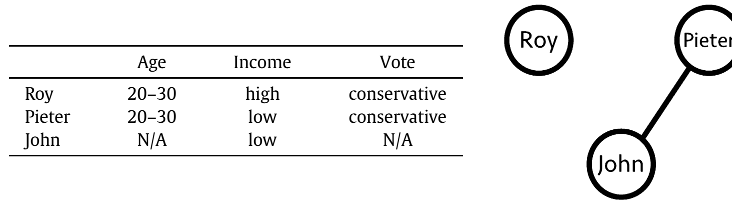
These candidate sets can be computed in ASP as follows:

$$\begin{aligned} add(f(I)) &\leftarrow not\ nAdd(f(I)),\ f(I). \\ nAdd(f(I)) &\leftarrow not\ add(f(I)),\ f(I). \\ remove(f(I)) &\leftarrow not\ nRemove(f(I)),\ f(I). \\ nRemove(f(I)) &\leftarrow not\ remove(f(I)),\ f(I). \end{aligned} \tag{5}$$

In fact, these rules introduce all the possible combinations of either adding or not adding every fact of F^{inc} to F_P, and either removing or not removing every fact of F^{inc} from F_P. By updating F_P correspondingly, all the possible candidate repairs are generated. In practice, the number of candidate repairs is usually high, and many valid repairs are found. Hence, a method to select a good repair is needed. A common strategy, which we refer to as the minimal repair approach, consists in finding repairs with the smallest number of modifications to F_P.

Definition 2. Minimal Repair

A minimal repair of an inconsistent answer set program P is a repair (A, B) such that for all other repairs (X, Y), it holds that $|A| + |B| \leq |X| + |Y|$.



	Age	Income	Vote
Roy	20–30	high	conservative
Pieter	20–30	low	conservative
John	N/A	low	N/A

Fig. 3. On the left, a table that shows the age, income class and vote of three people collected from a survey. On the right, a social network that represents friendship between people that took part in the survey. An edge between two people means that they are friends, otherwise, they are not.

While searching for a minimal repair is a reasonable strategy in absence of any background knowledge (supported by Occam's razor principle), our hypothesis is that we can find more accurate repairs by incorporating rules of thumb. While the nature of these rules of thumb is application dependent, we will assume that they can be encoded using ASP rules, and that these rules correspond to some cost-inducing program that does not interfere with the initial program. This allows us to describe the repair mechanisms based on these costs, independent of the specific choices of the rules of thumb. In other words, a rule of thumb is encoded as an ASP program which (i) does not derive any literals from the initial program P, and (ii) derives a cost value encoded by an integer, which will reflect a penalty value that the rule of thumb associates with a given repair. To use these rules of thumb, we create a new program Q based on the inconsistent program P and the rules of thumb, such that the answer sets of Q correspond to the optimal repairs of P. In order to do so, we extend P based on the rules in (5) that generate candidate repairs, and add the rules of thumb that encode a cost value for every candidate repair. We can then use these cost values, instead of the number of modifications to F_P, as the basis for selecting repairs.

Thus, the rules of thumb added to the ASP program act as "soft constraints". In fact, rules of thumb can encode different types of knowledge, including domain-free rules, i.e. rules that can be applied to a multitude of problems that are of similar structure (e.g. transitivity constraints in taxonomy problems), domain-specific properties (e.g. medical knowledge about cells found in the biological literature), and even user assumptions and intuitions. The cost values introduced by these rules can then be aggregated to define an overall preference ordering for the set of possible repairs. In Section 5 we will discuss several ways in which these costs can be aggregated.

Consider the following example to illustrate our approach. We have data about individuals collected from a survey: age group, income group, and vote. The age group is represented by intervals of 10 years (e.g. in20 represents 20–30, in30 represents 30–40, in40 represents 40–50, etc.), the income group is represented by three levels: high, medium and low, and the vote is either "liberal" or "conservative". Example data from the survey is shown in Fig. 3. We also have a social network that represents whether the individuals are friends (every node in the network graph corresponds to one person, and an edge between person A and person B reads "A and B are friends"). An example social network is shown in Fig. 3. The survey contains missing and/or wrong information. The goal is to correct wrong information, and possibly to complete missing information. In our example, we assume that it is known that "liberal" won the total vote. i.e. at least two out of the three persons voted "liberal". This problem can be encoded in ASP as follows:

$$\begin{array}{ll}
person(roy). & ageGrp(in20).\\
person(pieter). & ageGrp(in30).\\
person(john). & ageGrp(in40).\\
incomeGrp(low). & voteGrp(liberal).\\
incomeGrp(medium). & voteGrp(conservative).\\
incomeGrp(high). & income(roy, high).\\
age(roy, in20). & income(pieter, low).\\
age(pieter, in20). & income(john, low).\\
vote(roy, conservative). & friends(pieter, john).\\
vote(pieter, conservative). &
\end{array}$$

$$votesLib(T) \leftarrow T = \#count\{vote(X, liberal)\}.$$

$$\leftarrow votesLib(T),\ T < 2.$$

It is clear that this is an inconsistent ASP program, since the facts encode that Roy and Pieter voted "conservative", while the program also includes the constraint that there were at least two "liberal" votes. Let P be the program described above. In this example, we assume that the social network is correct, and that any inconsistencies directly result from errors in the survey data. In particular, we assume that $F^{inc} = \{age(roy, in20), age(pieter, in20), age(john, in20), age(roy, in30),$ $age(pieter, in30), age(john, in30), age(roy, in40), age(pieter, i\text{-}n40), age(john, in40), income(roy, high), income(roy, medium),$

income(*roy*, *lo-w*), *income*(*pieter*, *high*), *income*(*pieter*, *medium*), *income*(*pieter*, *low*), *income-*(*john*, *high*), *income*(*john*, *medium*), *income*(*john*, *low*), *vote*(*roy*, *conservativ-e*), *vote*(*roy*, *liberal*), *vote*(*pieter*, *conservative*), *vote*(*pieter*, *liberal*), *vote*(*john*, *-conservative*), *vote*(*john*, *liberal*)} and that $F^{con} = \{friends(pieter, john)\}$.

In order to find all the valid repairs, we first extend the program by introducing rules that generate all the possible candidate sets X and Y, as follows:

$$
\begin{aligned}
remove(age(P,A)) &\leftarrow age(P,A),\ not\ nRemove(age(P,A)).\\
nRemove(age(P,A)) &\leftarrow age(P,A),\ not\ remove(age(P,A)).\\
remove(income(P,I)) &\leftarrow income(P,I),\ not\ nRemove(income(P,I)).\\
nRemove(income(P,I)) &\leftarrow income(P,I),\ not\ remove(income(P,I)).\\
remove(vote(P,V)) &\leftarrow vote(P,V),\ not\ nRemove(vote(P,V)).\\
nRemove(vote(P,V)) &\leftarrow vote(P,V),\ not\ remove(vote(P,V)).\\
add(age(P,A)) &\leftarrow person(P),\ ageGrp(A),\ not\ age(P,A),\\
&\quad\ not\ nAdd(age(P,A)).\\
nAdd(age(P,A)) &\leftarrow person(P), ageGrp(A),\ not\ age(P,A),\\
&\quad\ not\ add(age(P,A)).\\
add(income(P,I)) &\leftarrow person(P),\ incomeGrp(I),\ not\ income(P,I),\\
&\quad\ not\ nAdd(income(P,I)).\\
nAdd(income(P,I)) &\leftarrow person(P),\ incomeGrp(I),\ not\ income(P,I),\\
&\quad\ not\ add(income(P,I)).\\
add(vote(P,V)) &\leftarrow person(P),\ voteGrp(V),\ not\ vote(P,V),\\
&\quad\ not\ nAdd(vote(P,V)).\\
nAdd(vote(P,V)) &\leftarrow person(P),\ voteGrp(V),\ not\ vote(P,V),\\
&\quad\ not\ add(vote(P,V)).
\end{aligned}
$$

Then, we add rules to update F_P with respect to the candidate repairs, as follows:

$$
\begin{aligned}
age_n(P,A) &\leftarrow age(P,A),\ not\ remove(age(P,A)).\\
age_n(P,A) &\leftarrow add(age(P,A)).\\
income_n(P,I) &\leftarrow income(P,I),\ not\ remove(income(P,I)).\\
income_n(P,I) &\leftarrow add(income(P,I)).\\
vote_n(P,V) &\leftarrow vote(P,V),\ not\ remove(vote(P,V)).\\
vote_n(P,V) &\leftarrow add(vote(P,V)).
\end{aligned}
$$

Note that the predicate $vote(X, liberal)$ is updated to $vote_n(X, liberal)$ in the rule "$votesLib(T) \leftarrow T = \#count\{vote(X, liberal)\}$". At this point, the answer sets of the program encode all the possible repairs of P. To solve the problem using the minimal repair method, we first add rules that compute the cost of adding and removing edges from F_P, as follows:

$$
\begin{aligned}
costAdding(X) &\leftarrow X = \#count\{add(F)\}.\\
costRemoving(Y) &\leftarrow Y = \#count\{remove(F)\}.\\
cost(0,C) &\leftarrow costAdding(X),\ costRemoving(Y), C = X + Y.
\end{aligned}
$$

Minimizing C in the predicate $cost(0, C)$ allows us to compute the minimal repairs. In this case, since we need at least two "liberal" votes, these repairs will consist in adding the fact *vote*(*john*, *liberal*), and changing at least one of the other two votes from "conservative" to "liberal". For example, changing Roy's vote from "conservative" to "liberal" corresponds to removing the fact *vote*(*roy*, *conservative*) from F_P, and then adding the fact *vote*(*roy*, *liberal*). Hence, changing a vote has a cost of 2. Since the minimal repair does not take into account any additional information, changing either Roy's vote or Pieter's vote to "liberal" are both valid solutions. From this condition alone, we get two possible minimal repairs that are equally valid, and it is up to the user to decide which repair to pick.

However, adding information in the form of rules of thumb allows us to choose who is most likely to have voted liberal. For example, let us consider the following four rules of thumb:

1. If two people have the same age, and are in the same income class, they are likely to have voted in the same way.
2. If two people are friends, they are likely to have voted in the same way.
3. If two people are friends, and they are in the same income class, they are likely to be in the same age group.
4. If a person is in the “high” income class, they are more likely to vote “conservative”.

These rules of thumb can be encoded in the program as follows:

$$penalty1(X,Y) \leftarrow age_n(X,A),\ age_n(Y,A),\ income_n(X,I),\ income_n(Y,I),$$
$$vote_n(X,V1),\ vote_n(Y,V2),\ V1 \mathrel{!=} V2.$$
$$cost(1,C) \leftarrow C = \#sum[penalty1(X,Y)].$$
$$penalty2(X,Y) \leftarrow friends(X,Y),\ vote_n(X,V1),\ vote_n(Y,V2),\ V1 \mathrel{!=} V2.$$
$$cost(2,C) \leftarrow C = \#sum[penalty2(X,Y)].$$
$$penalty3(X,Y) \leftarrow friends(X,Y),\ income_n(X,I),\ income_n(Y,I),$$
$$age_n(X,A1),\ age_n(Y,A2),\ A1 \mathrel{!=} A2.$$
$$cost(3,C) \leftarrow C = \#sum[penalty3(X,Y)].$$
$$penalty4(X) \leftarrow income_n(X,high),\ vote_n(X,liberal).$$
$$cost(4,C) \leftarrow C = \#sum[penalty4(X)].$$

For simplicity, we choose the penalty cost for every violation of a rule of thumb to be equal to 1. The simplest way to aggregate these penalty costs is to add the individual costs together with the cost of adding and removing facts from F_P using the following rule:

$$totalCost(T) \leftarrow T = \#sum[cost(X,C) = C].$$

Then, the repair that has the lowest total cost is considered the best repair. In this case, there is only one best repair and it satisfies all the rules of thumb. This repair has a total cost equal to 4. We get the best repair by removing the fact $vote(pieter, conservative)$, and adding the facts $vote(pieter, liberal)$, $vote(john, liberal)$, and $age(john, in20)$. Note that encoding additional knowledge using rules of thumb also allows us to fill out the missing age of John, which a minimal repair method does not provide.

5. Aggregating rules of thumb

While it is clear that carefully modelled additional knowledge should lead to better results, we are in particular interested in the case where only a straightforward encoding of rules of thumb is available. In the following, we describe several ways to aggregate the penalty costs corresponding to different rules of thumb and show how they can be implemented. Note that all repair methods take into consideration the penalties introduced by the rules of thumb as well as the penalty introduced by the repair operations (i.e. cost of adding and removing facts from F_P). The effectiveness of these repair methods for the case of GRNs is studied in Section 8.

5.1. Uniform weights approach

This is the simplest approach for using the rules of thumb without having access to training data. The idea is to choose the weights such that each rule of thumb has approximately the same impact on the choice of repair. To accomplish this, in each candidate solution (i.e., each candidate repair), we identify violations of each rule of thumb and increase the associated penalty cost by 1 for every violation. Then, we normalize the resulting penalty for each rule of thumb, such that the expected total cost associated with each rule of thumb is the same. The way in which this normalization factor is computed is problem-specific. In Section 8 we will explain how this can be done for the case of GRNs. Finally, the total cost of a repair is calculated by adding all the penalties together, along with the penalty associated with the repair operations, and the repair with the lowest total cost is considered the best repair. With this approach, even if there is a rule of thumb that can possibly be violated much more frequently than another one, both will have the same impact on the total cost of a repair.

Example 3. Suppose we want to create a map that displays the current temperatures of 100 cities in a specific region. In order to do so, we write a program that extracts a list of 100 temperature measurements for the corresponding cities from different weather websites. The goal is then to update the measurements that are likely wrong with more plausible values. Suppose we also have two rules of thumb R_1 and R_2. R_1 encodes that the temperature difference between cities in one region cannot be too large (i.e. all the measurements should belong to a certain temperature range). R_2 checks whether

every measurement was extracted from a trusted source. Naturally, the penalty from R_1 is either 0 if all the measurements belong to the same temperature range, or 1 otherwise. On the other hand, the penalty from R_2 is increased by 1 every time we find a measurement that was extracted from an untrusted source. In this case, since R_2 can potentially be violated 100 more times than R_1, we set the normalization factor of R_1 to 100, and the normalization factor of R_2 to 1. We keep the normalization factor of the penalty introduced by the repair operation (i.e. replacing a measurement by another) equal to 1, since we can perform at most 100 replacements.

Now suppose we have two repairs A and B. In repair A, the penalty from R_1 is equal to 1, and the penalty from R_2 is equal to 15. In repair B, the penalties introduced by R_1 and R_2 are 0 and 35 respectively. Additionally, repair A and repair B both required 10 repair operations. Hence, we consider two vectors $A = [1, 15, 10]$ and $B = [0, 35, 10]$. Note that the first two elements of each vector correspond to the penalties introduced by the rules of thumb, and the third element corresponds to the penalty introduced by the repair operations. We then apply the normalization factors on the penalties to calculate the total costs of every repair. For repair A, the total cost is $totalCost_A = (1 \times 100) + (15 \times 1) + (10 \times 1) = 115$ and for repair B, the total cost is $totalCost_B = (0 \times 100) + (35 \times 1) + (10 \times 1) = 45$. Since $45 < 115$, we prefer repair B over repair A using this method.

5.2. Violations improvements method

In contrast to the previous method, this method does not define a total ordering between repairs. In particular, here we consider a voting based approach, where a repair A is preferred over a repair B if for the majority of the rules of thumb there are more violations in B than in A.

Example 4. Suppose we have two repairs A and B, and three rules of thumb R_1, R_2 and R_3. In repair A, the penalty from R_1 is 30, the penalty from R_2 is 20, and the penalty from R_3 is 10. In repair B, the penalties introduced by the three rules of thumb are 25, 15 and 40 respectively. Both repairs required 8 repair operations each. Hence, we consider two vectors $A = [30, 20, 10, 8]$ and $B = [25, 15, 40, 8]$. For every rule of thumb, we check whether repair B improved on repair A based on the number of rule violations. For R_1, repair B improved on repair A because $25 < 30$. For R_2, repair B also improved on repair A because $15 < 20$. For R_3, repair B did not improve on repair A because $40 > 10$. The number of repair operations needed for both repairs is the same. Therefore, repair B is preferred over repair A using this method.

To find an optimal repair, we start by generating a random repair. Then, we use a second ASP program to find a repair that improves on the first repair, if such a better repair exists. This is done by generating a new repair, then comparing it with the old repair as described above, using simple rules that compare penalties and count the number of improvements. This process is continued, where each time we specifically look for repairs that improve on all the previously generated repairs, until no further improvements can be found. As the preference ordering in this case is not transitive, this is important to avoid Condorcet's paradox (i.e. a situation in which collective preferences can be cyclic), as illustrated in the following example.

Example 5. Assume we have a repair A, and two rules of thumb R_1 and R_2. The penalty from R_1 is 20 and the penalty from R_2 is 20, and the repair A requires 25 repair operations. We then consider the vector $A = [20, 20, 25]$. We may then generate a new repair B that requires 40 repair operations, and with the penalties from R_1 and R_2 equal to 10 and 10 respectively. The vector is then $B = [10, 10, 40]$, and B improves on A. After that, if we require an improvement from the previous repair only, we may generate the repair C that requires 30 repair operations, and with the penalties from R_1 and R_2 equal to 30 and 5 respectively $C = [30, 5, 30]$.

In Example 5, B improves on A, C improves on B, and A improves on C. For this reason, we impose the condition that a newly generated repair must improve on all the previously generated ones.

5.3. Leximin and leximax ordering

The idea of this aggregation method is based on the leximin and leximax orderings [51]. Let u be a vector of integers. We denote by u^* the vector defined by $u_i^* = u_{\gamma(i)}$ with γ a permutation of the components of u such that $u_{\gamma(1)} \leq u_{\gamma(2)} \leq \ldots \leq u_{\gamma(n)}$. The leximin ordering between two vectors u and v is then defined by $u > v$ iff $\exists k \leq n$ such that $u_k^* > v_k^*$ and $\forall i < k$, $u_i^* = v_i^*$. In other words, the leximin ordering looks to minimize the *smallest* value of every vector. If the smallest value of both vectors is identical, the second smallest is considered; if the second smallest value is also identical, the third smallest value is considered, etc. Similarly, the leximax ordering between two vectors u and v is defined by $u > v$ iff $\exists k \leq n$ such that $u_k^* > v_k^*$ and $\forall i > k$, $u_i^* = v_i^*$. In other words, the leximax ordering looks to minimize the *largest* value of every vector.

Example 6. Consider two vectors $A = [5, 1, 4, 8]$ and $B = [4, 6, 1, 7]$. We first sort the values in the vectors increasingly as $A_{inc} = [1, 4, 5, 8]$ and $B_{inc} = [1, 4, 6, 7]$. When using the leximin ordering, we will prefer A over B, as A_{inc} has a lower value

than B_{inc} in the first component where they differ (i.e. $5 < 6$). On the other hand, when using leximax, we sort the values in the vectors decreasingly as $A_{dec} = [8, 5, 4, 1]$ and $B_{dec} = [7, 6, 4, 1]$. Then, B will be preferred over A, as the highest values in B and A are respectively 7 and 8.

To implement the leximin ordering in ASP, we use #minimize statements with priorities, which are supported by the ASP solver *clasp*. We now illustrate this construction for the penalties from Example 6, for the case of leximin.

$$
\begin{aligned}
& a \mid b. \\
& cost(1,5) \leftarrow a. \\
& cost(2,1) \leftarrow a. \\
& cost(3,4) \leftarrow a. \\
& cost(4,8) \leftarrow a. \\
& cost(1,4) \leftarrow b. \\
& cost(2,6) \leftarrow b. \\
& cost(3,1) \leftarrow b. \\
& cost(4,7) \leftarrow b. \\
& rank(1,1) \mid rank(1,2) \mid rank(1,3) \mid rank(1,4). \\
& rank(2,1) \mid rank(2,2) \mid rank(2,3) \mid rank(2,4). \\
& rank(3,1) \mid rank(3,2) \mid rank(3,3) \mid rank(3,4). \\
& rank(4,1) \mid rank(4,2) \mid rank(4,3) \mid rank(4,4). \\
& \leftarrow rank(I,J),\ rank(I,K),\ J \mathrel{!}= K. \\
& \leftarrow rank(I,J),\ rank(E,J),\ I \mathrel{!}= E. \\
& \leftarrow rank(I,J),\ rank(I+1,L),\ cost(J,N), \\
& \quad cost(L,M),\ N > M. \\
& costRank1(C) \leftarrow rank(1,I),\ cost(I,C). \\
& costRank2(C) \leftarrow rank(2,I),\ cost(I,C). \\
& costRank3(C) \leftarrow rank(3,I),\ cost(I,C). \\
& costRank4(C) \leftarrow rank(4,I),\ cost(I,C). \\
& \#minimize[costRank1(C) = C\ @\ 4]. \\
& \#minimize[costRank2(C) = C\ @\ 3]. \\
& \#minimize[costRank3(C) = C\ @\ 2]. \\
& \#minimize[costRank4(C) = C\ @\ 1].
\end{aligned}
\tag{6}
$$

The literal "rank(I, J)" defines an ordering for every rule of thumb and reads: the index I is given to rule of thumb J. The rules of thumb are ordered from least violated to most violated using the three constraints. The optimization statement "#minimize[costRank1(C) = C @ 4]" reads: minimize the cost of the rule of thumb with rank 1 (i.e. the least violated rule of thumb) and give this operation the highest priority (priority = 4). In general, the four #minimize statements encode the fact that we want to minimize the number of violations, with the highest priority given to the least violated rule of thumb and the lowest priority given to the most violated rule of thumb.

The ASP code for leximax ordering is very similar, with the only difference that the sign ">" has been replaced by "<" in the rule:

$$\leftarrow rank(I,J),\ rank(I+1,L),\ cost(J,N),\ cost(L,M),\ N < M.$$

5.4. Z-score approach

The idea of this method is to look at the total penalty for each rule of thumb and compare this against the expected total penalty value, using the Z-score. This expected penalty is estimated based on a large number of repairs. In particular, given a sample S of repairs (how these repairs are sampled is discussed in the next paragraph), we generate for each repair s in S a vector $x^{(s)} = [x_1^{(s)}, x_2^{(s)}, \ldots, x_m^{(s)}]$ that contains the number of times $x_j^{(s)}$ each rule of thumb j ($j \in \{1, \ldots, m\}$) is violated. Then, we calculate the average number of times μ_j each rule of thumb was violated across all the repairs in the sample, as well as the standard deviation σ_j for the number of violations of every rule of thumb. Based on these averages and standard deviations, we then construct an answer set program for generating preferred repairs as follows. For each repair r, we count the number of violations of every rule of thumb $x^{(r)} = [x_1^{(r)}, x_2^{(r)}, \ldots, x_m^{(r)}]$ and use it to calculate the corresponding Z-score of every rule of thumb. The total cost of the repair r is then defined as the sum of these Z-scores:

$$\sum_{j=1}^{m} \text{Zscore}\left(x_j^{(r)}\right) = \sum_{j=1}^{m} \frac{x_j^{(r)} - \mu_j}{\sigma_j} \tag{7}$$

Comparing a repair with another then boils down to which repair has the smallest cost, i.e. smallest sum of Z-scores.

We consider four setups which differ in how we sample repairs for estimating the mean and variance. Variant A: we start with completely random repairs. These repairs have only one condition to satisfy, which is to remove the inconsistency in the ASP program. Variant B: we only consider minimal repairs in the sample. We generate these repairs using the minimal repair method discussed in Section 6. The assumption underlying this second setup is that minimal repairs will typically be more similar to the correct repairs than arbitrary repairs, and therefore the characteristics of minimal repairs might be more informative. Variant C: this variant consists of starting with a sample of repairs that are close to being minimal, e.g. repairs involving at most three times the number of operations in the minimal repairs. This represents a trade-off between the first two variants, acknowledging that while minimal repairs, on average, would be more plausible than arbitrary repairs, the correct repair is often not actually among the set of minimal repairs. Variant D: we consider repairs of any cost, but do not sample these repairs uniformly as in the first setup. In particular, since there are many more repairs with a high cost than repairs with a low cost, the sample in the first setup will be dominated by high-cost repairs. In this fourth setup, we therefore sample repairs such that the probability of selecting a repair with a given cost is uniform. In particular, we create groups of repairs of the same cost, ranging from a group of repairs with minimal cost to a group of repairs with maximal cost, and sample an equal number of repairs from each group.

6. Case study: gene regulatory networks

While some general principles can be derived, we argue that the best way to repair an inconsistent ASP program usually depends on domain-specific knowledge. In this paper we illustrate this for biological networks, an established application domain of ASP [28,27]. In this section, we briefly recall what GRNs are, and present a setup where inconsistencies arise. We show how to encode this setup in ASP, and recall the minimal repair method, a popular, straightforward method to resolve the inconsistencies.

A Gene Regulatory Network (GRN) is a network that represents the interactions between a group of cell genes. The nodes of a GRN are genes, whereas the edges of the network encode interactions between the genes. Two types of possible interactions between a pair of genes are usually considered: a gene either *activates* another gene or *inhibits* another gene. The intended meaning is that if gene A activates gene B, and A is active at time step t, then B becomes active at time step $t + 1$. Likewise, if gene A inhibits gene B, and A is active at time step t, then B becomes inactive at time step $t + 1$. In the case where a gene is activated and inhibited simultaneously, different activation rules may be applied to determine the subsequent state [52]. Different kinds of experimental observations can be used to automatically construct GRNs, such as using a sparse Gaussian Markov Random Field, which relates network topology with the covariance observed in the gene measurements [12], and the concept of specificity-determining residues [13].

We consider a setup that consists of the following. We have an automatically generated GRN describing cell interactions. Since it has been automatically constructed, it is likely to be imperfect, in the sense that we may later obtain observations that are inconsistent with the behavior predicted by the network. In particular, we have a time-series table that lists the state of the genes in this GRN at consecutive time steps. At every time step, a given gene can either be *active* or *inactive*. This table represents experimental observations, and is generally used to answer a wide range of biological questions about the corresponding genes [53]. The table we have at our disposal represents data from new experiments that was not available during the GRN generation process. Our problem then boils down to repairing the GRN, if necessary, such that it becomes consistent with the data from the time-series table, i.e. such that the GRN can correctly predict the evolution of the states in the table.

6.1. Encoding GRNs in ASP

We recall how a GRN and corresponding time-series table can be encoded in ASP, as presented in [28,54,52,27]. This includes the observed time-series table data, as well as the GRN graph that might be inconsistent with the table. For every

gene X, we introduce the fact $gene(X)$. For every edge from gene X to gene Y, we introduce the fact $edgeInit(X, Y, 1)$ if X activates Y, or $edgeInit(X, Y, -1)$ if X inhibits Y. To encode the time-series table, we include facts of the form $active(X, T)$ and $inactive(X, T)$ which indicate that gene X is active at time T and that gene X is inactive at time T respectively. We also represent every time step with the fact $time(T)$ with $0 \leq T \leq t_{\text{final}}$, with t_{final} representing the final time step observed.

Then, to check for consistency between the graph and the table, three types of rules are needed. First, we need activation and inhibition rules for the graph, which determine whether a gene is activated or inhibited (or neither) at each time step. We use the following activation rule as described in [52]: if a gene is positively regulated by at least one other gene, and it is not negatively regulated by any other gene, then it is activated. A similar rule is used to determine when a gene is inhibited, as shown in the following:

$$\begin{aligned} receivesAct(Y, T) &\leftarrow activates(X, Y),\ active(X, T). \\ receivesInh(Y, T) &\leftarrow inhibits(X, Y),\ active(X, T). \\ activated(Y, T) &\leftarrow receivesAct(Y, T-1),\ not\ receivesInh(Y, T-1). \\ inhibited(Y, T) &\leftarrow receivesInh(Y, T-1),\ not\ receivesAct(Y, T-1). \end{aligned} \tag{8}$$

Second, we need rules to determine the state of every gene at every time step based on the aforementioned activation and inhibition interactions given by the GRN graph, and on the gene states at the previous time step given by the table:

$$\begin{aligned} inactive(Y, T) &\leftarrow active(Y, T-1),\ inhibited(Y, T). \\ active(Y, T) &\leftarrow active(Y, T-1),\ not\ inhibited(Y, T). \\ active(Y, T) &\leftarrow inactive(Y, T-1),\ activated(Y, T). \\ inactive(Y, T) &\leftarrow inactive(Y, T-1),\ not\ activated(Y, T). \end{aligned} \tag{9}$$

Third, we add the following rule to check whether the states of the genes generated by the activation and inhibition rules of the graph correspond with the states of the genes in the time-series table

$$\leftarrow active(Y, T),\ inactive(Y, T). \tag{10}$$

It is clear that the resulting ASP program is consistent if and only if the GRN graph is compatible with the time-series table. Otherwise, the program is inconsistent.

6.2. Repairing GRNs using minimal repair

Several methods have already been developed that use ASP to repair inconsistencies found in GRNs [5,28,54]. These methods usually consist in finding some kind of minimal repair. In the following, we will discuss in more detail how a minimal repair method can be encoded in ASP, i.e. we want to construct an ASP program whose answer sets are the minimal repairs of the inconsistent program. We are interested in a setting where we want to restore consistency by adding, removing or changing facts (while leaving other rules unchanged). Note that we can focus on modifying facts without loss of generality. Indeed, if we want to make the rule $\alpha \leftarrow \beta$ optional, we can always write this rule as $\alpha \leftarrow \beta, x$ and add x as a fact. Then, removing the fact x essentially means removing the rule $\alpha \leftarrow \beta$.

Our program for finding minimal repairs is based on the one proposed in [5], with the main differences being the cause of inconsistency and hence the consistency check, as well as the repair operations proposed. As possible repair operations in our case study, we consider either adding or removing an edge between two genes. Thus, there are four possibilities for every pair of nodes: add a new activation edge, add a new inhibition edge, remove an existing edge or do nothing. We encode the first three options using the literals $addActEdge(U, V)$, $addInhEdge(U, V)$ and $removeEdge(U, V, S)$ respectively in the following rules:

$$\begin{aligned} edge(U, V) &\leftarrow edgeInit(U, V, S). \\ addActEdge(U, V) &\leftarrow gene(U),\ gene(V),\ not\ edge(U, V), \\ &\qquad not\ addInhEdge(U, V),\ not\ nAddActEdge(U, V). \\ nAddActEdge(U, V) &\leftarrow gene(U),\ gene(V), not\ edge(U, V), \\ &\qquad not\ addInhEdge(U, V),\ not\ addActEdge(U, V). \\ addInhEdge(U, V) &\leftarrow gene(U),\ gene(V),\ not\ edge(U, V), \\ &\qquad not\ addActEdge(U, V),\ not\ nAddInhEdge(U, V). \\ nAddInhEdge(U, V) &\leftarrow gene(U),\ gene(V),\ not\ edge(U, V), \\ &\qquad not\ addActEdge(U, V),\ not\ addInhEdge(U, V). \end{aligned}$$

$$\begin{aligned} removeEdge(U,V,S) &\leftarrow edgeInit(U,V,S),\ not\ nRemoveEdge(U,V,S). \\ nRemoveEdge(U,V,S) &\leftarrow edgeInit(U,V,S),\ not\ removeEdge(U,V,S). \end{aligned} \tag{11}$$

These rules ensure that the new literals representing the possible repair operations are optional, and introduce the constraint that at most one of these literals can be made true for a given pair of edges. Then, to take the generated repair into account, we consider the following rules:

$$\begin{aligned} activates(U,V) &\leftarrow edgeInit(U,V,1),\ not\ removeEdge(U,V,1). \\ activates(U,V) &\leftarrow addActEdge(U,V). \\ inhibits(U,V) &\leftarrow edgeInit(U,V,-1),\ not\ removeEdge(U,V,-1). \\ inhibits(U,V) &\leftarrow addInhEdge(U,V). \end{aligned} \tag{12}$$

The ASP program consisting of the rules (8)–(12) has one answer set for each possible repair of the original GRN, i.e. each modification that will make it consistent with the table. To restrict the answer sets to those that correspond to minimal repairs only, we first define the cost of a repair, using the following rules:

$$\begin{aligned} addEdge(U,V,1) &\leftarrow addActEdge(U,V). \\ addEdge(U,V,-1) &\leftarrow addInhEdge(U,V). \\ costAdding(X) &\leftarrow X = \#count\{addEdge(U,V,S)\}. \\ costRemoving(Y) &\leftarrow Y = \#count\{removeEdge(U,V,S)\}. \\ repairCost(Z) &\leftarrow costAdding(X),\ costRemoving(Y),\ Z = X+Y. \\ &\#minimize[repairCost(Z) = Z]. \end{aligned} \tag{13}$$

These rules contain *aggregates* and *optimization statements* supported by the ASP solver *clasp*, which behave like built-in functions in the solver. For example, in (13), the aggregate *#count* intuitively counts the number of instances of the literals *addEdge* and *removeEdge* that are true, and stores the results in variables X and Y respectively. The optimization statement *#minimize* acts as a function that finds the answer sets with the lowest value held by the variable Z in the literal *repairCost(Z)*. This restricts the answer sets to those that represent the repairs with the lowest cost.

7. Rules of thumb for repairing GRNs

The flexibility of ASP allows us to encode different kinds of properties as rules of thumb. In particular, combining negation-as-failure with the aggregates provided by most ASP solvers creates a powerful framework for the user to express background knowledge about the considered domain. In the following, we will show how we can encode rules of thumb about GRNs which are found in the biological literature.

7.1. Rule 1: last time step as fixed state

In [55], it is stated that every gene network converges to a final stable state. We can use this biological knowledge by checking whether the GRN we are trying to repair indeed converges to the stable state indicated by the final time step in the table. To implement this, we consider an additional time step $t_{\text{final}+1}$ after the last time step mentioned in the table. This rule of thumb is satisfied if and only if the state of the network remains constant in this last time step. We can easily check whether this is the case using rules similar to (8)–(10). If the repair is still correct, i.e. if the graph is still consistent with the table after the addition of the time step $t_{\text{final}+1}$, then the time step t_{final} is indeed a fixed state.

7.2. Rule 2: degree of a gene

In [8], Kauffman found that a genetic network will behave chaotically unless there is a restriction on the number of regulatory inputs and outputs per node. This can be encoded as a rule of thumb by putting bounds on the degree of each node.

First, we need to find the degree of every node in the GRN graph, given by $k = k_{\text{in}} + k_{\text{out}}$ with k_{in} being the number of incoming edges and k_{out} the number of outgoing edges of the node. We then need to verify whether these degrees fall within a certain range. We explain how we choose this range in Section 8. We then use the predicate *kBadGene* to encode the number of genes whose degree falls outside the range limits. The penalty introduced by this rule of thumb is increased for every "bad gene" found.

The following ASP rules are used to count this number of bad genes:

$$
\begin{aligned}
edgeAfterRepair(U,V) &\leftarrow activates(U,V).\\
edgeAfterRepair(U,V) &\leftarrow inhibits(U,V).\\
kOut(C,X) &\leftarrow X = \#count\{edgeAfterRepair(C,D)\},\ gene(C).\\
kIn(C,X) &\leftarrow X = \#count\{edgeAfterRepair(D,C)\},\ gene(C).\\
kDegree(C,Z) &\leftarrow kIn(C,X),\ kOut(C,Y),\ Z = X+Y.\\
kBadGene(C) &\leftarrow kDegree(C,Z),\ Z < k_{\min}.\\
kBadGene(C) &\leftarrow kDegree(C,Z),\ Z > k_{\max}.\\
kBadGenes(X) &\leftarrow X = \#count\{kBadGene(C)\}.\\
cost(2,C) &\leftarrow kBadGenes(C).
\end{aligned} \tag{14}
$$

7.3. Rule 3: total number of edges

Another rule of thumb can be derived from the fact that various biological properties in a gene network depend on the number of non-zero interactions between the nodes of this network, as is discussed in [56]. This leads us to impose the constraint that similar gene networks would more likely have a similar number of total interactions.

To encode this rule of thumb, we count the total number of interactions between the genes of the GRN, and check whether this number falls within a certain range that is derived from similar GRNs (see Section 8). This rule of thumb is satisfied if and only if the number of interactions is inside the range. The ASP rules to encode this rule of thumb are similar to (14).

7.4. Rule 4: likely interactions based on gene state

In [56], it is observed that nodes tend to be positively regulated by nodes that are active at earlier states of a cell cycle and negatively regulated by nodes that are active later in the process. To take this observation into account, we divide the genes into likely activators and likely inhibitors based on whether they are active during the first half or the second half of the time-series table respectively. Note that the same gene can be both a likely activator and a likely inhibitor. We then check the outgoing edges of every gene, and increase the cost of the repair every time a likely activator, which is not also a likely inhibitor, inhibits another gene, or a likely inhibitor, which is not also a likely activator, activates another gene. The penalty is increased for every "bad edge" that is found.

We use the following ASP rules to compute the corresponding penalty:

$$
\begin{aligned}
likelyAct(C) &\leftarrow active(C,T),\ T <= t_{\text{half}}.\\
likelyInh(C) &\leftarrow active(C,T),\ T > t_{\text{half}}.\\
badEdge(C,D) &\leftarrow likelyAct(C),\ inhibits(C,D),\ not\ likelyInh(C),\ C\ != D.\\
badEdge(C,D) &\leftarrow likelyInh(C),\ activates(C,D),\ not\ likelyAct(C),\ C\ != D.\\
badEdges(X) &\leftarrow X = \#count\{badEdge(C,D)\}.\\
cost(4,C) &\leftarrow badEdges(C).
\end{aligned} \tag{15}
$$

7.5. Rule 5: network diameter

In [57], it is stated that the diameter (i.e. the length of the shortest path between the two nodes that are furthest apart in the network) of GRN graphs tends to be very small. To encode this knowledge, we first need to make sure that every gene of the network is reachable, using the following rules:

$$
\begin{aligned}
link(X,Y) &\leftarrow edgeAfterRepair(X,Y),\ X\ != Y.\\
link(Y,X) &\leftarrow edgeAfterRepair(X,Y),\ Y\ != X.\\
reachable(X) &\leftarrow link(1,X).\\
reachable(Y) &\leftarrow reachable(X),\ link(X,Y).\\
&\leftarrow gene(X),\ not\ reachable(X).
\end{aligned} \tag{16}
$$

Then, we find the shortest distance between every pair of genes by finding all the possible paths between them, and minimizing the number of path links. The longest of these shortest distances is the diameter of the network.[1]

$$
\begin{aligned}
dist(X,Y,1) &\leftarrow link(X,Y),\ X\,!=Y.\\
dist(X,Y,2) &\leftarrow link(X,A),\ link(A,Y),\ X\,!=Y.\\
dist(X,Y,3) &\leftarrow link(X,A),\ link(A,B),\ link(B,Y),\ X\,!=Y.\\
&\dots\\
smallestDist(X,Y,D) &\leftarrow D = \#min[dist(X,Y,C)=C],\ dist(X,Y,Z).\\
diameter(D) &\leftarrow D = \#max[smallestDist(X,Y,C)=C].
\end{aligned}
\tag{17}
$$

The penalty associated with this rule of thumb depends on whether the diameter that was found falls within a certain range limit (see Section 8).

7.6. Rule 6: dominant motifs

A motif is a small pattern with usually three or four nodes that is found repeatedly in a network graph. In [58], the idea of dominant motifs in GRNs is discussed. Inspired by this observation, we will consider a rule of thumb that is based on how many dominant motifs are shared between a given repaired GRN and the GRNs of similar cell cycles. This allows us to encode that similar networks are likely to share the same dominant motifs.

To implement this rule of thumb, we first use an external program described in [59] to find the dominant motifs of popular GRNs in the literature. The GRN that we are repairing is not used during this step. We then encode these motifs in our ASP program and try to maximize the number of times they appear in the repaired network. The penalty introduced by this rule should be smaller when more dominant motifs are found. Therefore, we start with a penalty which is equal to a maximum penalty value, and then count the number of dominant motifs in the repaired network. For every instance of a dominant motif that we find, we decrease the penalty of this rule of thumb by 1.

7.7. Rule 7: size of basin of attractors

In [60], it is stated that the size of the basin of attractors (i.e. the number of stable states to which most initial states of the network converge) in a GRN is a vital quantity in terms of understanding network behavior and may relate to other network properties such as stability. This allows us to check whether the state of the repaired network with the largest basin size indeed corresponds to the final stable state in the time-series table.

This final rule of thumb is not implemented in ASP, but is instead used as an additional filtering step after the ASP program has been used to find good repairs. To apply this rule, we need to find the final state corresponding to every possible initial state of a network, using a standalone program described in [61]. We then need to make sure that the most frequently occurring final state of the network indeed corresponds to its state at the final time step t_{final} given by the time-series table. We apply this method for every repaired network (i.e. for every answer set that we get from our ASP program), and filter out the ones that do not satisfy this property. Note that we do not implement this rule of thumb in ASP due to the requirement of using an external program to find all possible final states of a network. However, this final rule of thumb still allows us to apply a filter on the computed answer sets, thus improving the repair process.

8. Experimental results

To compare the effectiveness of the proposed aggregation methods, consider the following biological scenario: a draft of a GRN that represents the relations between some genes was generated, but the GRN graph is not consistent with experimental observations. These observations consist of a time-series table that describes the state of the studied genes at different time steps of the corresponding cell cycle. In our experiments, we use the following 5 GRNs (see Table 2): *Arabidopsis* [62], *Budding Yeast* [63], *C. Elegans* [64], *Fission Yeast* [65], and *Mammalian Cell Cycle* [66]. We also have 5 time-series tables that are consistent with these GRNs. It is important to note that Boolean networks constitute a simple model to represent GRNs. However, this model is valuable towards obtaining information about the network topology, and is more precise that previously thought [62]. Note that typically there are many Boolean networks that are compatible with a given time series table. In fact, we verified that there are millions of Boolean networks that are compatible with the time series data that we use in our experiments.

To recreate the biological scenario that was previously described, we corrupt every GRN by adding and removing edges. Our methods then aim to repair the corrupted GRNs. Every time we corrupt a network, we remove R randomly chosen edges,

[1] The rules that calculate the distance between a pair of nodes could be encoded in a more elegant version, namely using the rule $dist(X,Y,D) \leftarrow dist(X,A,L),\ link(A,Y),\ D = L+1$. However, this version is too slow when performing experiments, so a more straightforward encoding is used.

Table 1
N and R values for every GRN in every corruption setup.

GRN	Number of edges	Corruption Setup ID									
		1	2	3	4	5	6	7	8	9	10
Arabidopsis	22	N = 1 R = 2	N = 2 R = 1	N = 2 R = 2	N = 4 R = 8	N = 7 R = 7	N = 11 R = 6	N = 7 R = 16	N = 9 R = 14	N = 14 R = 9	N = 18 R = 5
Budding Yeast	34	N = 1 R = 2	N = 3 R = 2	N = 3 R = 3	N = 6 R = 12	N = 11 R = 11	N = 17 R = 9	N = 11 R = 24	N = 14 R = 21	N = 21 R = 14	N = 28 R = 7
C. Elegans	21	N = 1 R = 2	N = 2 R = 1	N = 2 R = 2	N = 4 R = 8	N = 7 R = 7	N = 11 R = 6	N = 7 R = 15	N = 9 R = 13	N = 13 R = 9	N = 17 R = 5
Fission Yeast	25	N = 1 R = 2	N = 2 R = 1	N = 2 R = 2	N = 4 R = 9	N = 8 R = 8	N = 13 R = 7	N = 8 R = 18	N = 10 R = 16	N = 16 R = 10	N = 20 R = 6
Mammalian	39	N = 1 R = 2	N = 3 R = 2	N = 4 R = 3	N = 6 R = 14	N = 12 R = 12	N = 20 R = 10	N = 12 R = 28	N = 16 R = 24	N = 24 R = 16	N = 32 R = 8

Table 2
Characteristics of the GRN graphs.

GRN	Number of nodes	Number of edges	Edges-nodes ratio	Average degree	Diamete
Arabidopsis	10	22	2.2	4.4	4
Budding Yeast	11	34	3.1	6.2	3
C. Elegans	8	21	2.6	5.3	3
Fission Yeast	9	25	2.8	5.6	3
Mammalian	10	39	3.9	7.8	3

and subsequently add N randomly chosen edges (choosing between activation and inhibition edges with equal probability). We set N and R as percentages of the initial number of edges for each network that we are corrupting. For our experiments, we consider 10 corruption setups by varying the percentages N and R in the following way: N = 1%/R = 5%, N = 6%/R = 3%, N = 8%/R = 6%, N = 15%/R = 35%, N = 30%/R = 30%, N = 50%/R = 25%, N = 30%/R = 70%, N = 40%/R = 60%, N = 60%/R = 40%, and N = 80%/R = 20%. The values of N and R for every GRN in every corruption setup are shown in Table 1. These setups vary from light corruptions (setup 1–3) to heavier corruptions (setup 4–10). Note that, in a real biological setting, both light and heavy corruption may be expected. "Indeed, inferring gene networks from experimental observations is a daunting task" [67]. The methods used to predict gene interactions based on data from time-series tables give varying results with very different predicted networks (e.g. DREAM challenges[2]). In particular, in the DREAM4 challenge, the applicants were asked to infer gene interactions based on given time-series data. The leaderboards of this challenge[3] show that the top scoring methods predicted gene networks with edge counts varying from 10 edges to 54 edges, with the best scoring network having an edge count of 18. Therefore, generated networks may vary significantly in the number of edges from the actual network that they are trying to predict.

Every time we select a network to corrupt and repair, we learn the relevant parameters of the rules of thumb from the other four, uncorrupted networks. For Rule 2, we learn the degrees k_{min} and k_{max} from the other four networks by setting k_{min} as the smallest average degree value of the other four networks and k_{max} as the largest average degree value. The range [$diameter_{min}$, $diameter_{max}$] in Rule 5 is learned similarly, where $diameter_{min}$ is the smallest diameter value of the other four networks, and $diameter_{max}$ is the largest diameter value. For Rule 3, the range of the total number of edges is calculated as follows. We learn from the other four networks the ratio of number of edges per node, and we keep the minimum ($ratio_{min}$) and maximum ($ratio_{max}$) values that we find. Then, we determine what the expected number of edges should be for the test network by multiplying these two ratios with the number of nodes. Table 2 contains all the relevant characteristics of every GRN that are needed to calculate the aforementioned parameters.

When repairing using the uniform weights approach, we set the maximum penalty for each rule of thumb to be equal to the total number of initial edges of the GRN that we are trying to repair. We chose this number because it represents the largest possible penalty that a rule of thumb can introduce if we increase the cost by one for every violation (Rule 4 in particular). Based on this value, we introduce a normalization factor F for every rule of thumb. Let "EdgesNum" and "GenesNum" be respectively the number of edges and the number of genes in the GRN that we are trying to repair. In Rule 1 (last time step as fixed state), the rule can be violated at most once (the last time step is either a fixed state or not a fixed state). Therefore, the penalty of this rule is either equal to 0 in the case where it is satisfied, or otherwise equal to 1. The normalization factor for this rule is then F = EdgesNum. In Rule 2 (degree of a gene), every gene in the network can possibly be considered as a "bad gene" hence violating this rule and increasing its penalty cost by 1. Then, to normalize the total penalty, we multiply it by the ratio EdgesNum/GenesNum. This gives us F = EdgesNum/GenesNum. In this way, we make sure that the maximum penalty is equal to EdgesNum (if all the genes of the network are "bad genes",

[2] http://dreamchallenges.org/.
[3] https://www.synapse.org/#!Synapse:syn2825304/wiki/71132.

Table 3
The normalization factor (F) for every rule of thumb used by the uniform weights approach.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7
Normalization factor (F)	EdgesNum	$\frac{\text{GenesNum}}{\text{EdgesNum}}$	1	1	EdgesNum	EdgesNum	N/A[a]

[a] Rule 7 is used as a final filtering step and is not implemented in ASP. Therefore, a normalization factor is not applicable.

Table 4
Average number of repair operations applied to a GRN at all corruption setups by the minimal repair method and the 3 best proposed aggregation methods (rounded to the nearest integer).

GRN	Minimal repair method	Uniform weights method	Variant C of Z-score approach	Variant D of Z-score approach
Arabidopsis	16	22	28	25
Budding Yeast	27	32	33	31
C. Elegans	14	23	25	27
Fission Yeast	19	24	29	26
Mammalian	35	42	48	44

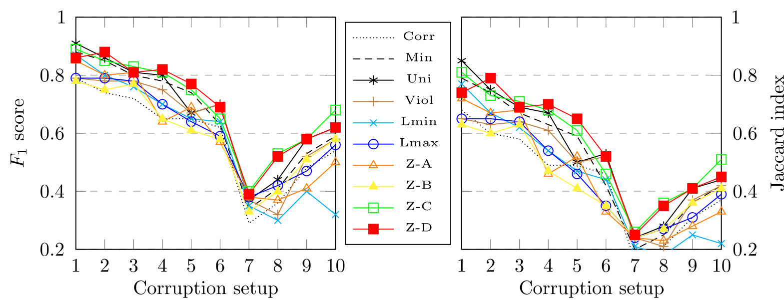


Fig. 4. F_1 score and Jaccard index of Arabidopsis network.

the maximum penalty is $\text{penalty}_{max} = (\text{GenesNum}) \times (\text{EdgesNum}/\text{GenesNum}) = \text{EdgesNum}$). We follow the same reasoning for all the other rules of thumb in our case study. The normalization factors for every rule of thumb used by the uniform weights approach can be found in Table 3.

To evaluate our results, we use the F_1 score and Jaccard index which we calculate as follows. Let A be the set of edges of the original network and B the set of edges of the repaired network. We write $|A|$ and $|B|$ for the number of edges of the original and repaired network respectively. The F_1 score is given by $F_1 = 2 \times (\text{precision} \times \text{recall}) \ / \ (\text{precision} + \text{recall})$, with $\text{precision} = |A \cap B| \ / \ |B|$ and $\text{recall} = |A \cap B| \ / \ |A|$. The Jaccard index is given by $J(A, B) = |A \cap B| \ / \ |A \cup B|$. We use the F_1 score because it is a standard measure of accuracy that considers both precision and recall, and the Jaccard index because it is a standard measure of similarity between sets. We run every experiment (i.e. every corruption setup on every network) 10 times and report the average F_1 score and Jaccard Index of the best repair that was found. In the case where multiple repairs with the same minimum cost were found, we select the first repair that we get from the solver as best repair. To run our experiments, we used the ASP grounder *gringo* and the ASP solver *clasp* running on a 2.7 GHz Intel Core i5 CPU with 8 GB of RAM. Despite the large search space, all the aggregation methods have fast run times, varying between a few seconds to at most 5 minutes, depending on the corruption setup. Our experiment setups and results can be found at: http://www.cwi.ugent.be/RepairInconsistentASP.html.

The results of our experiments are shown in Figs. 4–8. Every figure corresponds to one of the 5 GRNs, and represents the F_1 score and Jaccard index for every corruption setup. Every graph shows the results for 9 different repair methods: the minimal repair method (Min), the uniform weights approach (Uni), the Violations Improvements method (Viol), leximin ordering (Lmin), leximax ordering (Lmax), and variants A-D of the Z-score approach (Z-A, Z-B, Z-C, Z-D). The average F_1 score and Jaccard index of the corrupted network (Corr) is also shown. The average graphs for F_1 score and Jaccard index for all the GRNs are shown in Fig. 9 and Fig. 10 respectively.

Based on these graphs, we can conclude that the Z-score approach is the best method to aggregate the rules of thumb in the absence of training data. In particular, variant C of this approach, in which the sampled repairs are close to being minimal (Section 5.4) outperforms all the other aggregation methods, as well as the minimal repair approach. In fact, we notice a consistent improvement in both metrics when repairing the GRNs using this method, compared to all the other methods. Additionally, this method shows the most significant increase in F_1 score and Jaccard index between the corrupted and repaired GRNs. We notice from the graphs that most standard aggregation methods i.e. the leximin and leximax ordering methods, and the violation improvements method perform poorly compared to the minimal repair method for some GRNs, while showing better results for other GRNs. These varying results render these methods unreliable, and a key reason for this is presumably the lack of suitable weights. The Z-score approach addresses this issue by learning weights from unsupervised

Table 5
F_1 score of every repair method on every GRN for the setup where a generated GRN happens to be consistent with the time-series table, but different than the original ground-truth network.

	Initial	Min	Uni	Viol	Lmin	Lmax	Z-A	Z-B	Z-C	Z-D
Arabidopsis	0.32	0.32	0.34	0.31	0.32	0.30	0.36	0.31	0.36	0.35
Budding Yeast	0.83	0.83	0.87	0.84	0.82	0.83	0.84	0.81	0.91	0.87
C. Elegans	0.55	0.55	0.59	0.53	0.52	0.50	0.53	0.51	0.61	0.62
Fission Yeast	0.56	0.56	0.57	0.48	0.50	0.48	0.42	0.42	0.61	0.58
Mammalian	0.35	0.35	0.38	0.35	0.35	0.32	0.33	0.30	0.40	0.36

Table 6
Jaccard index of every repair method on every GRN for the setup where a generated GRN happens to be consistent with the time-series table, but different than the original ground-truth network.

	Initial	Min	Uni	Viol	Lmin	Lmax	Z-A	Z-B	Z-C	Z-D
Arabidopsis	0.22	0.22	0.24	0.20	0.22	0.19	0.26	0.21	0.27	0.24
Budding Yeast	0.75	0.75	0.77	0.75	0.73	0.74	0.76	0.73	0.82	0.77
C. Elegans	0.38	0.38	0.42	0.35	0.33	0.31	0.36	0.34	0.45	0.46
Fission Yeast	0.39	0.39	0.40	0.32	0.33	0.32	0.27	0.26	0.45	0.42
Mammalian	0.24	0.24	0.27	0.24	0.24	0.20	0.21	0.19	0.30	0.26

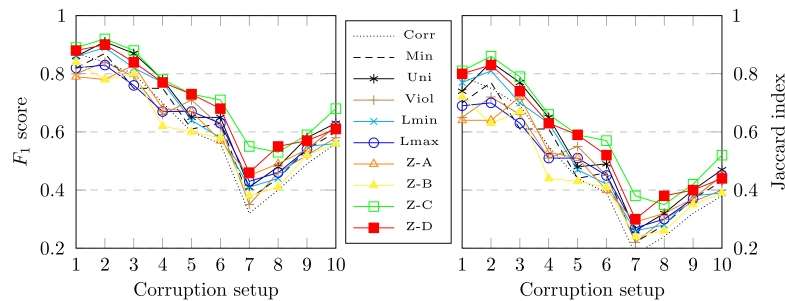


Fig. 5. F_1 score and Jaccard index of Budding Yeast network.

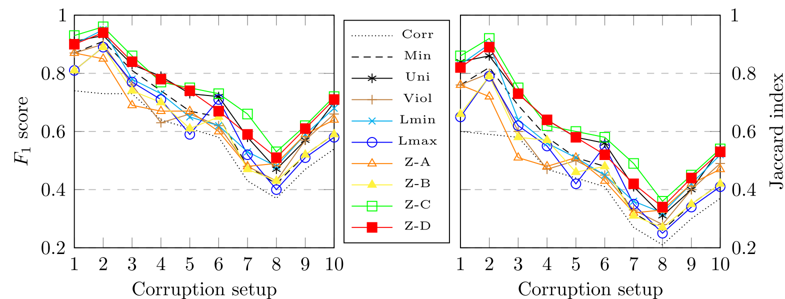


Fig. 6. F_1 score and Jaccard index of C. Elegans network.

data (viz. the repairs that we generate automatically). Furthermore, the way these repairs have been automatically generated gives us interesting insight into the expected costs of the best repairs. The fact that variant C of the Z-score approach is the best performing method shows us that the best repairs have a total cost that is relatively close to the minimal repair cost.

Additionally, there are two methods that also outperform the minimal repair approach: the uniform weights method (Section 5.1) and variant D of the Z-score approach (Section 5.4). In the absence of training data, setting uniform weights to all the rules of thumb works remarkably well. This confirms the conclusions in [68], which state that when predicting a numerical criterion, equal-weighting models tend to outperform average random models. It is important to note that these methods are not simply selecting the most promising among the minimal repairs (instead of a random minimal repair), but that the optimal repairs they find are indeed not minimal. Table 4 shows the average number of repair operations applied to a corrupted GRN by the aforementioned best repair methods. It is clear that all three approaches make more changes to a network than the minimal repair method does, confirming that while using minimal repairs is convenient, more optimal repairs are not necessarily minimal.

Finally, we considered one extra setup where a generated network happens to be consistent with the time-series table, but different than the original network. For this setup, instead of taking the original, ground-truth network and corrupting

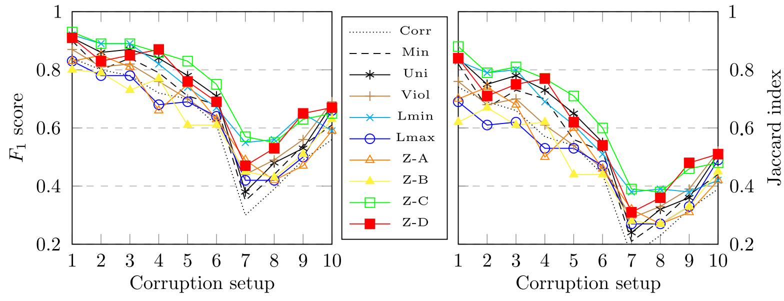


Fig. 7. F_1 score and Jaccard index of Fission Yeast network.

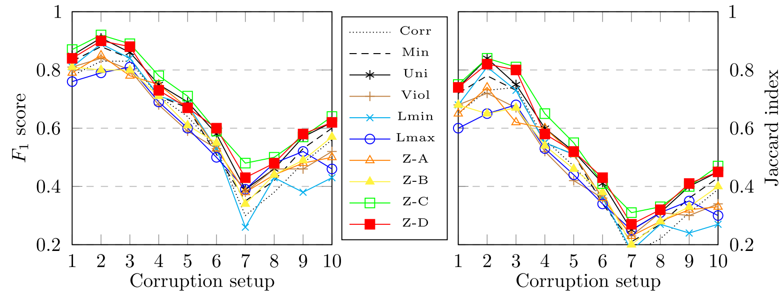


Fig. 8. F_1 score and Jaccard index of Mammalian network.

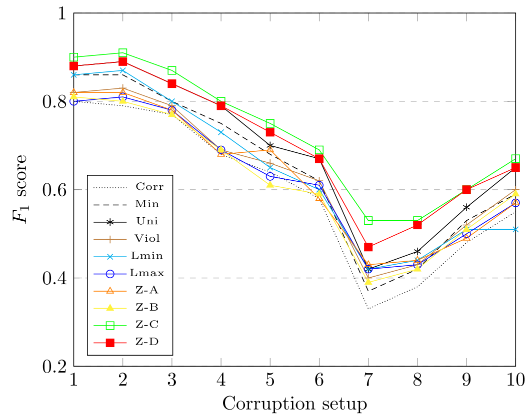


Fig. 9. Average graph for F_1 score.

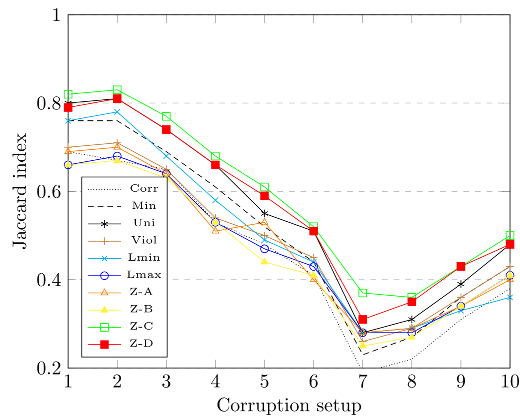


Fig. 10. Average graph for Jaccard index.

it by removing and adding edges, we randomly generate a new network from scratch that is consistent with the given time-series table. Then, we apply our repair methods to it to reduce the violations of the rules of thumb. Similar to our previous setups, we run this experiment 10 times, and report the average F_1 score and Jaccard index of the best repair that was found. The F_1 scores of the corrupted network and of the resulting best repairs for every repair method are shown in Table 5, and their corresponding Jaccard indices are shown in Table 6. Note that in this setup, the minimal repair method simply returns the randomly generated network, since there are no repair operations to be made. The results of this experiment confirm our previous findings that the uniform weights approach, variant C of the Z-score approach and variant D of the Z-score approach improve on both the F_1 score and the Jaccard index in comparison with the minimal repair method.

9. Conclusion

Answer set programming (ASP) is steadily gaining traction as a declarative programming language for modeling and simulating systems. As with all knowledge representation formalisms, correctly capturing domain knowledge into ASP programs is a challenging task. Errors in the encoding can result in ASP programs that have no answer sets. A common technique to restore the consistency of such inconsistent ASP programs is the minimal repair method. Supported by Occam's razor principle, this technique adds or removes the smallest number of facts to ensure that the program has at least one answer set. While searching for a minimal repair is certainly reasonable in absence of any background knowledge, in this paper we showed that more accurate repairs can be found by incorporating a small number of domain specific rules of thumb, or soft constraints, on what desired repairs should look like.

The technique we proposed in this paper makes very modest assumptions about the available domain knowledge. Instead of relying on carefully modelled expert knowledge, our approach only requires a straightforward encoding of ideas from the domain literature. In addition – unlike for instance existing approaches based on Markov logic – our technique does not require manual tuning of the importance weights of the rules of thumb, nor learning such weights from training data. To aggregate the impact of different rules of thumb in choosing the best repair, we have proposed the use of 5 techniques: (1) a *uniform weights approach* in which all rules have the same impact on the total cost of a repair; (2) a *violations improvements method* that favors repair A over repair B if A outperforms B for the majority of the rules of thumb; (3) a *leximin ordering approach* that favors repair A over repair B if A does really well on satisfying one of the rules of thumb, i.e. better than B does on any of the rules of thumb; (4) a *leximax ordering approach* that favors repair A over repair B if B does really poorly on satisfying one of the rules of thumb, i.e. worse than A does on any of the rules of thumb; (5) a *Z-score approach* that compares the number of violations against an expected number.

To validate our approach, we have applied it in a case study for simulating the behavior of Gene Regulatory Networks (GRNs). We corrupted known GRNs in various ways and then attempted to restore them with our methods, using 7 rules of thumb from the biological literature. We have found that the variant of the Z-score approach that uses automatically generated repairs that are close to being minimal outperforms all the other repair methods in terms of F_1 score and Jaccard index, including the minimal repair approach. Our results have also confirmed that these methods are not simply selecting the most promising among the minimal repairs, but the optimal repairs that they produce are indeed not minimal. In fact, our best method shows that the most plausible repairs are close to being minimal.

In future work, it would be interesting to see whether the Z-score approach can be improved by using more advanced methods for outlier detection. For instance, one-class Support Vector Machines (SVMs) could be trained from randomly sampled repairs, using feature vectors that contain the number of violations of each repair, which in principle would allow us to take into account how different rules of thumb interact.

Acknowledgements

We acknowledge the Ghent University Multidisciplinary Research Partnership 'Bioinformatics: From Nucleotides to Networks (N2N)' project [01MR0410], and ERC Starting Grant [637277]. This research is funded by FWO Flanders (fund for scientific research) project [G035612N].

References

[1] V. Lifschitz, What is answer set programming?, in: AAAI, vol. 8, 2008, pp. 1594–1597.
[2] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: ICLP/SLP, vol. 88, 1988, pp. 1070–1080.
[3] V. Lifschitz, Action languages, answer sets, and planning, in: The Logic Programming Paradigm, Springer, 1999, pp. 357–373.
[4] V. Lifschitz, Answer set programming and plan generation, Artif. Intell. 138 (2002) 39–54.
[5] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, P. Veber, Repair and prediction (under inconsistency) in large biological networks with answer set programming, in: Twelfth International Conference on the Principles of Knowledge Representation and Reasoning, 2010.
[6] M. Arenas, L. Bertossi, J. Chomicki, Answer sets for consistent query answering in inconsistent databases, Theory Pract. Log. Program. 3 (2003) 393–424.
[7] O. Arieli, M. Denecker, B. Van Nuffelen, M. Bruynooghe, Database repair by signed formulae, in: International Symposium on Foundations of Information and Knowledge Systems, Springer, 2004, pp. 14–30.
[8] S.A. Kauffman, The Origins of Order: Self-Organization and Selection in Evolution, Oxford University Press, 1993.
[9] H. De Jong, Modeling and simulation of genetic regulatory systems: a literature review, J. Comput. Biol. 9 (2002) 67–103.
[10] T. Chen, V. Filkov, S.S. Skiena, Identifying gene regulatory networks from experimental data, Parallel Comput. 27 (2001) 141–162.

[11] I. Shmulevich, E.R. Dougherty, W. Zhang, From Boolean to probabilistic Boolean networks as models of genetic regulatory networks, Proc. IEEE 90 (2002) 1778–1792.
[12] P. Menéndez, Y.A. Kourmpetis, C.J. ter Braak, F.A. van Eeuwijk, Gene regulatory networks from multifactorial perturbations using graphical lasso: application to the dream4 challenge, PLoS ONE 5 (12) (2010) e14147.
[13] J.J. Ellis, B. Kobe, Predicting protein kinase specificity: Predikin update and performance in the dream4 challenge, PLoS ONE 6 (7) (2011) e21169.
[14] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, N. Leone, Team-building with answer set programming in the gioia-tauro seaport, Theory Pract. Log. Program. 12 (2012) 361–381.
[15] A.M. Smith, M. Mateas, Answer set programming for procedural content generation: a design space approach, IEEE Trans. Comput. Intell. AI Games 3 (2011) 187–200.
[16] H. Erdogan, O. Bodenreider, E. Erdem, Finding semantic inconsistencies in umls using answer set programming, in: AAAI, 2010.
[17] E. Merhej, S. Schockaert, M. De Cock, Using rules of thumb for repairing inconsistent answer set programs, in: Scalable Uncertainty Management, Springer, 2015, pp. 368–381.
[18] T. Eiter, Data integration and answer set programming, in: Logic Programming and Nonmonotonic Reasoning, Springer, 2005, pp. 13–25.
[19] T. Zhu, Z. Zhang, Y. Zhai, Z. Gao, A processing method for inconsistent answer set programs based on minimal principle, in: International Conference on Automatic Control and Artificial Intelligence (ACAI 2012), IET, 2012, pp. 270–274.
[20] T. Syrjänen, Debugging inconsistent answer set programs, in: Proc. NMR, vol. 6, 2006, pp. 77–83.
[21] K. Marple, G. Gupta, Dynamic consistency checking in goal-directed answer set programming, arXiv:1405.3603.
[22] M. Brain, M. De Vos, Debugging logic programs under the answer set semantics, in: Answer Set Programming, 2005.
[23] M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, S. Woltran, Debugging asp programs by means of asp, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, 2007, pp. 31–43.
[24] J.P. Delgrande, T. Schaub, H. Tompits, S. Woltran, et al., Belief revision of logic programs under answer set semantics, in: Proceedings, Eleventh International Conference on Principles of Knowledge Representation and Reasoning, vol. 8, 2008, pp. 411–421.
[25] T. Eiter, M. Fink, J. Moura, Paracoherent answer set programming, in: Twelfth International Conference on the Principles of Knowledge Representation and Reasoning, 2010, pp. 486–496.
[26] S. Dworschak, S. Grell, V.J. Nikiforova, T. Schaub, J. Selbig, Modeling biological networks by action languages via answer set programming, Constraints 13 (2008) 21–65.
[27] T. Fayruzov, J. Janssen, D. Vermeir, C. Cornelis, Modelling gene and protein regulatory networks with answer set programming, Int. J. Data Mining Bioinf. 5 (2011) 209–229.
[28] M. Gebser, A. König, T. Schaub, S. Thiele, P. Veber, The bioasp library: asp solutions for systems biology, in: ICTAI, vol. 1, 2010, pp. 383–389.
[29] M. Gebser, T. Schaub, S. Thiele, P. Veber, Detecting inconsistencies in large biological networks with answer set programming, Theory Pract. Log. Program. 11 (2011) 323–360.
[30] M. Richardson, P. Domingos, Markov logic networks, Mach. Learn. 62 (2006) 107–136.
[31] D. Lowd, P. Domingos, Efficient weight learning for Markov logic networks, in: Knowledge Discovery in Databases: PKDD 2007, Springer, 2007, pp. 200–211.
[32] T.N. Huynh, R.J. Mooney, Max-margin weight learning for Markov logic networks, in: Machine Learning and Knowledge Discovery in Databases, Springer, 2009, pp. 564–579.
[33] Z. Sun, Z. Wei, J. Wang, H. Hao, Scalable learning for structure in Markov logic networks, in: Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence, 2014.
[34] J. Huber, C. Meilicke, H. Stuckenschmidt, Applying Markov logic for debugging probabilistic temporal knowledge bases, in: Proceedings of the 4th Workshop on Automated Knowledge Base Construction (AKBC), 2014.
[35] S. Ghosh, N. Shankar, S. Owre, Machine reading using Markov logic networks for collective probabilistic inference, in: Proceedings of ECML-CoLISD, 2011.
[36] S. Schoenmackers, O. Etzioni, D.S. Weld, J. Davis, Learning first-order horn clauses from web text, in: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2010, pp. 1088–1098.
[37] R. Mooney, Relational learning of pattern-match rules for information extraction, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, vol. 334, 1999.
[38] S. Riedel, L. Yao, A. McCallum, Modeling relations and their mentions without labeled text, in: Machine Learning and Knowledge Discovery in Databases, Springer, 2010, pp. 148–163.
[39] M. Nickles, A. Mileo, Probabilistic inductive logic programming based on answer set programming, arXiv:1405.0720.
[40] K. Bauters, S. Schockaert, M. De Cock, D. Vermeir, Possible and Necessary Answer Sets of Possibilistic Answer Set Programs, Tools with Artificial Intelligence, vol. 1, IEEE, 2012, pp. 836–843.
[41] P. Nicolas, L. Garcia, I. Stéphan, C. Lefèvre, Possibilistic uncertainty handling for answer set programming, Ann. Math. Artif. Intell. 47 (2006) 139–181.
[42] J.C. Nieves, M. Osorio, U. Cortés, Semantics for possibilistic disjunctive programs, in: Logic Programming and Nonmonotonic Reasoning, Springer, 2007, pp. 315–320.
[43] P. Nicolas, L. Garcia, I. Stéphan, A possibilistic inconsistency handling in answer set programming, in: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, Springer, 2005, pp. 402–414.
[44] J. Lee, Y. Wang, A probabilistic extension of the stable model semantics, in: AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning, 2015, pp. 96–102.
[45] G. Brewka, T. Eiter, Preferred answer sets for extended logic programs, Artif. Intell. 109 (1999) 297–356.
[46] G. Brewka, Complex preferences for answer set optimization, in: Proceedings of KR, 2004, pp. 213–223.
[47] T. Eiter, W. Faber, N. Leone, G. Pfeifer, Computing preferred answer sets by meta-interpretation in answer set programming, Theory Pract. Log. Program. 3 (2003) 463–498.
[48] C. Zepeda, M. Osorio, J.C. Nieves, C. Solnon, D. Sol, Applications of preferences using answer set programming, in: Answer Set Programming, Citeseer, 2005.
[49] S. Costantini, A. Formisano, Modeling preferences and conditional preferences on resource consumption and production in asp, J. Algorithms 64 (2009) 3–15.
[50] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Clasp: a conflict-driven answer set solver, in: Logic Programming and Nonmonotonic Reasoning, Springer, 2007, pp. 260–265.
[51] D. Dubois, H. Fargier, H. Prade, Beyond min aggregation in multicriteria decision: (ordered) weighted min, discri-min, leximin, in: The Ordered Weighted Averaging Operators, Springer, 1997, pp. 181–192.
[52] M. Mushthofa, G. Torres, Y. Van de Peer, K. Marchal, M. De Cock, Asp-g: an asp-based method for finding attractors in genetic regulatory networks, Bioinformatics (2014), btu481.
[53] Z. Bar-Joseph, Analyzing time series gene expression data, Bioinformatics 20 (2004) 2493–2503.
[54] N. Mobilia, A. Rocca, S. Chorlton, E. Fanchon, L. Trilling, Logical modeling and analysis of regulatory genetic networks in a non monotonic framework, in: Bioinformatics and Biomedical Engineering, Springer, 2015, pp. 599–612.

[55] F. Li, T. Long, Y. Lu, Q. Ouyang, C. Tang, The yeast cell-cycle network is robustly designed, Proc. Natl. Acad. Sci. USA 101 (2004) 4781–4786.
[56] K.-Y. Lau, S. Ganguli, C. Tang, Function constrains network architecture and dynamics: a case study on the yeast cell cycle boolean network, Phys. Rev. E 75 (2007) 051907.
[57] R. Cohen, S. Havlin, Scale-free networks are ultrasmall, Phys. Rev. Lett. 90 (2003) 058701.
[58] T.I. Lee, N.J. Rinaldi, F. Robert, D.T. Odom, Z. Bar-Joseph, G.K. Gerber, N.M. Hannett, C.T. Harbison, C.M. Thompson, I. Simon, et al., Transcriptional regulatory networks in saccharomyces cerevisiae, Science 298 (2002) 799–804.
[59] S. Wernicke, F. Rasche, Fanmod: a tool for fast network motif detection, Bioinformatics 22 (2006) 1152–1153.
[60] L. Yang, Y. Meng, C. Bao, W. Liu, C. Ma, A. Li, Z. Xuan, G. Shan, Y. Jia, Robustness and backbone motif of a cancer network regulated by mir-17-92 cluster during the g1/s transition, PLoS ONE 8 (3) (2013) e57009.
[61] N. Berntenis, M. Ebeling, Detection of attractors of large Boolean networks via exhaustive enumeration of appropriate subspaces of the state space, BMC Bioinform. 14 (2013) 361.
[62] S. Bornholdt, Boolean network models of cellular regulation: prospects and limitations, J. R. Soc. Interface 5 (2008) S85–S94.
[63] R.F. Hashimoto, H. Stagni, C.H. Higa, Budding yeast cell cycle modeled by context-sensitive probabilistic Boolean network, in: IEEE International Workshop on Genomic Signal Processing and Statistics, in: GENSIPS 2009, IEEE, 2009, pp. 1–4.
[64] X. Huang, L. Chen, H. Chim, L.L.H. Chan, Z. Zhao, H. Yan, Boolean genetic network model for the control of c. elegans early embryonic cell cycles, Biomed. Eng. Online 12 (2013).
[65] M.I. Davidich, S. Bornholdt, Boolean network model predicts cell cycle sequence of fission yeast, PLoS ONE 3 (2) (2008) e1672.
[66] A. Fauré, A. Naldi, C. Chaouiya, D. Thieffry, Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle, Bioinformatics 22 (2006) e124–e131.
[67] A. Baralla, W.I. Mentzen, A. De La Fuente, Inferring gene networks: dream or nightmare?, Ann. N.Y. Acad. Sci. 1158 (2009) 246–256.
[68] R.M. Dawes, The robust beauty of improper linear models in decision making, Am. Psychol. 34 (1979) 571.