Universiteit Leiden

Master Computer Science

Programming a Stochastic Constraint Optimisation Algorithm, by Optimisation

Name:	Daniël Fokkinga
Student ID:	1532960
Date:	6/12/2019
Specialisation:	Advanced Data Analytics
1st supervisor:	Prof. dr. Holger H. Hoos
2nd supervisor:	Anna Louise D. Latour Msc.
3nd supervisor:	Marie Anastacio Msc.
2nd reader:	Prof. dr. Siegfried Nijssen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Stochastic Constraint Optimisation Problems (SCOPs), such as the *viral marketing problem* and *powergrid reliability problem*, arise in fields like industry, governance and science. Recent developments combine techniques from the fields of *Probabilistic Logic Programming* and *Constraint Programming*, and make it possible to model and solve such SCOPs efficiently. Solving SCOPs exactly is NP-hard, and to solve real-world problems, the design of exact SCOP solving methods must be highly optimised. To optimise the design of a recently developed solving method for SCOPs on monotonic probability distributions, we follow the principle of *Programming by Optimisation*. We expose design choices and add alternatives to those choices. This allows us to optimise these choices using *Automated Algorithm Configuration*. We compare the performance of our optimised SCOP solver to that of an expert-chosen default configuration of the solver. For a set of viral marketing problems, the optimised solver runs up to 51 times faster and solves more than 80% of the instances that could not be solved within a cutoff time of ten minutes by the default. For a set of powergrid reliability problems, the optimised solver solves 9% more instances overall, and solves instances up to 17 times faster.

Contents

Chapter 1

Introduction

In fields like industry, governance and science, problems in which one has to make optimal decisions under constraints and uncertainty are common.

Consider for example the *viral marketing problem*, a well-known problem in the data mining literature [28]. This problem is defined on a probabilistic network, where nodes correspond to people and the directed edges to stochastic influence relationships, indicating how likely a person is to influence another. We want to leverage *word-of-mouth marketing* to promote a new product in the network. To start this process, we are given k free samples to distribute to people in the network. Which group of k people is the most influential?

Another example is the *powergrid reliability problem* [16]. This problem is defined on a powergrid, where nodes correspond to consumers and producers of power, and edges to powerlines. In the event of a natural disaster, like an earthquake or hurricane, powerlines might break. If too many of them do, consumers may become disconnected from the grid and lose power. Each powerline has a certain probability of remaining intact during a disaster. By reinforcing powerlines we can increase this probability. This can be expensive and there is a limited budget for powerline maintenance. Which powerlines do we reinforce such that we maximise the expected number of power consumers that are still connected to a power producer after a disaster, while not exceeding our budget?

We call constraint optimisation problems that involve a constraint or objective function with a stochastic component *Stochastic Constraint Optimisation Problems (SCOPs)* [29]. Besides the two problems mentioned above, other examples of SCOPs include the problem of *signaling-regulatory pathway inference* [13] or a variant of the *landscape connectivity problem* [45].

A recently developed method leverages modelling and solving techniques from the fields of *Constraint Programming (CP)* and *Probabilistic Logic Programming (PLP)* to solve SCOPs on monotonic probability distributions exactly [31]. This method consists of three stages. The first two stages use PLP for modelling the problem as a logic program and compiling its probability distributions to a data structure that supports tractable probabilistic inference. The third stage uses techniques from CP to search for an optimal solution, in a way that takes advantage of specific properties of this data structure.

While this method has shown its merit in a proof of principle, it has not been optimised yet. Solving a problem such as the viral marketing problem, is in general an NP-hard task [28]. The optimisation of the solving algorithm is important, if we want the algorithm to scale to large problems.

The proposed solution to this problem is two-fold. First, many design choices are hard-coded in the current solver. We apply the principle of *Programming by Optimisation (PbO)* [21] by exposing those choices as parameters and providing alternatives to these choices. Then, we apply *Automated Algorithm Configuration (AAC)* [20] to find which combination of those design choices performs best on a given set of problems.

Our contributions are the following:

1. We apply PbO to the pipeline presented by Latour *et al.* [31]: we expose parameters for configuration and implement alternative design choices, before using AAC to optimise the resulting solver;
2. we show that the automatically configured version of this SCOP solver outperforms the hand-configured version presented earlier by Latour *et al.* [31].

The remainder of this thesis is organised as follows. In Chapter 2 we provide the definition of the SCOPs we study, along with examples and we show why they are challenging to solve. We then present an extensive outline of the method we use the solve them, followed by an introduction to PbO and AAC. In Chapter 5 we list the different parameters and design choices we consider in this work. We present experimental results in Chapter 6 and we conclude in Chapter 7, where we also provide some thoughts on future work.

Chapter 2

Stochastic Constraint Optimisation Problems

Problems such as the viral marketing problem and the powergrid reliability problem can be modelled as a *Stochastic Constraint Optimisation Problem (SCOP)*. In this chapter, we show how to model these problems as a SCOP and describe a naïve approach for solving them. We also discuss why this approach does not scale well and therefore, what makes SCOPs such challenging problems.

2.1 Modelling SCOPs

To describe how to model SCOPs, we begin with a general introduction to SCOPs and further illustrate this using two examples that are both defined on a probabilistic network.

2.1.1 General SCOP model

In this thesis, we consider SCOPs that are defined for two different types of Boolean variables. They can have the value *True* or *False*. A variable to which a value can be assigned is a *decision variable*: their values are determined by choice. A variable for which the value depends on a probability is a *stochastic variable*: each stochastic variable has its own independent probability that determines its value. An assignment of truth values to a set of decision variables is called a *strategy*, σ.

The objective is to maximise the *objective function*:

$$\sum_i \rho_i \cdot v_i, \tag{2.1}$$

where v_i can be either the value assigned to decision variable i or a conditional probability $P(\phi_i \mid \sigma)$; this conditional probability represents the probability of an event ϕ_i happening, given a strategy σ. The meaning of ϕ_i is further specified depending on the context of the SCOP under consideration, it could for example be a stochastic variable taking a specific value. Later in this chapter we give some examples of such events. With each v_i we associate a reward $\rho_i \in \mathbb{R}^+$, such that the objective function represents *expected utility*. In this thesis, we always assume $\rho_i = 1$ for simplicity. Generalizing the approaches discussed in this thesis to solve problems where $\rho_i \neq 1$, is trivial as it only requires to multiply the conditional probabilities with the appropriate reward to calculate the value of the objective function.

In practice, multiple different events can happen in a single problem and we are often only interested in a subset of events. We call this subset the *set of interest*, Φ.

To solve an SCOP a strategy has to be found that maximises the objective function and satisfies a *constraint*. In this thesis, we consider linear constraints on the *cardinality* of the solution [38]: the number of decision variables that are *True* in the strategy σ. To find a solution that satisfies this constraint and maximises the objective function, we can convert the objective function from Equation (2.1) into a *stochastic constraint*.

$$\sum_i \rho_i \cdot v_i > \theta \tag{2.2}$$

This results in a *constraint satisfaction* problem. By repeatedly solving this problem, while increasing the threshold $\theta \in \mathbb{R}^+$ until no more solution is found, we can find the maximum value for the function from Equation (2.1). The threshold θ takes the best value for $\sum_i \rho_i \cdot v_i$ found so far for a valid solution and thus increases each time we solve the constraint satisfaction problem.

The SCOPs we consider in this thesis have the special property that the probability distributions that are involved in the objective function are *monotonic*. This means that if we change a strategy σ such that additional decision variables are *True*, the value for Equation (2.1) can never decrease. In other words, for each conditional probability, $P(\phi \mid \sigma) \geq P(\phi \mid \sigma')$ holds for all σ' and σ where σ' is equal to σ except for one or more decision variable that are *True* in σ' and *False* in σ. This condition holds for the examples described in Section 2.1.2 and Section 2.1.3.

2.1.2 The viral marketing problem

In the viral marketing problem, the goal is to maximise the expected number of people that buy our product. By leveraging *word-of-mouth marketing* we can make people that buy our product turn their acquaintances into new buyers. Specifically, we assume stochastic relationships between people that determine how likely they are to be influenced by other specific people in their social network. To initiate the word-of-mouth process, we can distribute a limited number of free samples to people in the network. We want to give our samples to the most influential people, but how do we select this group? For simplicity we make two assumptions in the following example:

- A person that receives a free sample will always buy the product.
- If a person u buys the product and influences v, v will also buy the product.

Example 2.1.1. Modelling a viral marketing problem. We formulate a viral marketing problem for the

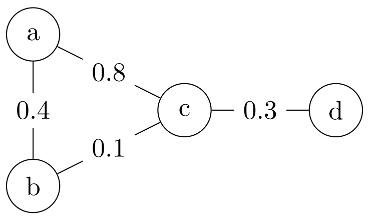


Figure 2.1: Social network of four persons $\{a, b, c, d\}$, edges represent the mutual stochastic trust relationships between the people [31].

network in Figure 2.1 as follows. For every node i, we define a decision variable $d_i \in \{0, 1\}$, and for every edge (i, j) a stochastic variable t_{ij} with a probability of evaluating to *True* that is equal to the label on edge (i, j). We represent the event that person i buys our product with ϕ_i.

Suppose that the people in this network can be divided in multiple categories and we are only interested in turning people that belong to a specific category into customers. A possible reason for this could be that we are only selling our product in a specific location, then we want to maximise the expected number of people in that location that buy our product. We model this by defining our set of interest to be, e.g., $\Phi = \{\phi_a, \phi_b\}$, meaning we are only interested in the events that persons a and b buy our product.

The objective is to find a strategy σ that maximises $\sum_{\phi \in \Phi} P(\phi \mid \sigma)$ (the objective function).

Finally, we constrain the number of people that can receive a free sample: $\sum_{i \in \{a,b,c,d\}} d_i \leq k$, where $k \in \mathbb{N}^+$ is the number of free samples that are available for distribution.

2.1.3 The powergrid reliability problem

In the powergrid reliability problem we are given a network in which nodes represent *power producers* (such as powerplants), *power consumers* or *intermediate grid nodes*. The nodes are connected by *powerlines*. We want to maximise the expected number of consumers that are still connected to at least one power producer in the case of a natural disaster, during which powerlines can break. Each powerline has a probability that it remains intact during a natural disaster. By reinforcing a powerline, we can increase this probability. We are given a budget for such reinforcements. Which powerlines do we reinforce such that we maximise the number of consumers that are still connected to producers after a natural disaster, while respecting our budget?

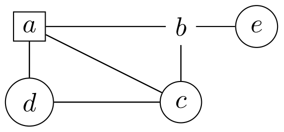


Figure 2.2: Network of powerlines between a producer a, three consumers $\{c, d, e\}$ and an intermediate grid node b.

Example 2.1.2. Modelling a powergrid reliability problem. For the network in Figure 2.2, we model this problem as follows. For every powerline l we define a decision variable d_l that indicates if the powerline is chosen to be reinforced, and a stochastic variable t_l, which indicates if the powerline remains intact during a disaster. In our model, the probability p_l that t_l is *True* is defined as

$$p_l = \begin{cases} p_{l,1} & \text{if } d_l = \mathit{False}; \\ p_{l,2} & \text{otherwise,} \end{cases}$$

with $p_{l,1} \leq p_{l,2}$. We represent the event that a power consumer i is still connected to at least one power producer as ϕ_i.

We define a set of interest, e.g., $\Phi = \{\phi_d, \phi_e\}$, if we are only interested in a specific subset of power consumers. A reason for this could be that we are only interested in buildings that fall under a certain category to be connected to a power producer. For example if we want to maximise the expected number of hospitals that are still connected.

The objective is to find a strategy σ that maximises $\sum_{\phi \in \Phi} P(\phi \mid \sigma)$.

We put a constraint on the powerlines that we reinforce: $\sum_{l \in \text{lines}} d_l \cdot \gamma_l \leq \beta$, where $\beta \in \mathbb{R}^+$ is our budget, and $\gamma_l \in \mathbb{R}^+$ is the cost for reinforcing powerline l.

Note that in this problem, the decision variables are associated with the edges in the network. In contrast, for the viral marketing problem, the decision variables are associated with the nodes from the network.

2.2 Naïve approach to solving SCOPs

To solve any SCOP, we need a method to evaluate a strategy σ. To be able to do this, we first have to define the conditional probability $P(\phi \mid \sigma)$ for any σ. For this purpose we can use *Weighted Model Counting (WMC)* [10], we represent an event ϕ with a weighted propositional formula over decision variables and stochastic variables.

Example 2.2.1. WMC for a viral marketing problem. Consider the viral marketing problem from Section 2.1.2. We model the event that person d buys the product by the following formula

$$\phi_d = d_d \vee (d_c \wedge t_{cd}) \vee (d_b \wedge t_{bc} \wedge t_{cd}) \vee (d_a \wedge t_{ac} \wedge t_{cd}) \vee (d_b \wedge t_{ba} \wedge t_{bc} \wedge t_{cb}) \vee (d_a \wedge t_{ab} \wedge t_{bc} \vee t_{cd}) \tag{2.3}$$

where each stochastic variable t_{ij} has a weight $p(t_{ij}) \in [0, 1]$ corresponding to the probability of the variable evaluating to *True*.

A possible strategy for $k = 1$ is to give a free sample only to person a. This results in a σ where d_a is *True* and the other decision variables are *False*. Given σ, we calculate $P(\phi_d \mid \sigma)$ by the sum of all possible *logical models* of the formula for ϕ_d. A logical model in this context is a combination of truth values for each stochastic variable such that the formula evaluates to *True*. One example of a model for ϕ_d given $\sigma = \{d_a = \top, d_b = d_c = d_d = \bot\}$ is $\{t_{ac} = t_{cd} = \top, t_{ab} = t_{bc} = \bot\}$. The probability that the stochastic variables take the values in this model is $p(t_{ac}) \cdot p(t_{cd}) \cdot (1 - p(t_{ab})) \cdot (1 - p(t_{bc})) = 0.8 \cdot 0.3 \cdot (1 - 0.4) \cdot (1 - 0.1) = 0.1296$.

To find the optimal strategy using this WMC approach, we can enumerate every possible strategy. The number of possible strategies is 2^n for n decision variables, therefore this is infeasible for large n. The other issue with solving SCOPs comes from the difficulty of performing WMC, which is a #P-complete problem in the general case [39]. To evaluate a single strategy, we have to sum the probabilities of all the possible logical models.

Table 2.1: Possible models for ϕ_d given $\sigma = (d_a = \top, d_b = d_c = d_d = \bot)$

Truth values				
t_{ab}	t_{ac}	t_{bc}	t_{cd}	$P(\phi_d \mid \sigma)$
$\bot$	$\top$	$\bot$	$\top$	$(1-0.4)\cdot 0.8\cdot(1-0.1)\cdot 0.3 = 0.1296$
$\bot$	$\top$	$\top$	$\top$	$(1-0.4)\cdot 0.8\cdot 0.9\cdot 0.3 = 0.0144$
$\top$	$\bot$	$\top$	$\top$	$0.4\cdot(1-0.8)\cdot 0.1\cdot 0.3 = 0.0024$
$\top$	$\top$	$\bot$	$\top$	$0.4\cdot 0.8\cdot(1-0.9)\cdot 0.3 = 0.0864$
$\top$	$\top$	$\top$	$\top$	$0.4\cdot 0.8\cdot 0.1\cdot 0.3 = 0.0096$ +
				0.2424

Using the viral marketing example we can illustrate why it is difficult to calculate the probability for a single strategy. To calculate $P(\phi_d \mid \sigma)$ for the strategy σ where only d_a is *True*, we have to calculate the probability that there is a *at least* one path from node a to node d in the network from Figure 2.1. Such a path represents the possibility that person d ends up being turned into a customer following the free sample handed out to person a. Thus, to evaluate a strategy, we have to sum the probabilities of all possible models that at least one path exists. However, simply calculating the sum of all these probabilities is not possible because, in general, these probabilities are not mutually independent. It is therefore important to take into account that there might be an overlap between different paths and we have to be careful not to count the same model more than once. In the case of our example, the given strategy results in five possible models, see Table 2.1. Each of these combinations of truth values for the stochastic variables results in ϕ_d evaluating to *True* given σ.

For larger, more complex networks, calculating the sum of all possible models for each strategy becomes increasingly expensive. This is a problem when using WMC to solve SCOPs such as the viral marketing problem and powergrid reliability problem. In the following chapter we introduce a recently developed method that is designed to tackle the issues with this approach and efficiently solve SCOPs.

Chapter 3

Solving SCOPs

In Chapter 2 we describe the difficulties of solving SCOPs exactly, specifically the difficulty related to WMC and the large search space of possible strategies. In earlier work [31], a solving method is proposed that tackles these difficulties and is shown to solve SCOPs in an efficient manner. This method, which we will refer to as SCOP SOLVER, consists of three stages: modelling, compilation and solving. In this chapter we describe and discuss these stages.

3.1 Modelling stage

In Section 2.2 we describe the approach of performing WMC on a weighted propositional formula ϕ. To construct such a formula ϕ from the original SCOP definition, various methods exists [14,16]. SCOP SOLVER uses *Probabilistic Logic Programming* in the *modelling stage* to model the SCOP with a probabilistic programming language. In this thesis and in the context of the previous work on SCOP SOLVER, we study SCOPs that are defined on probabilistic networks (similar to Examples 2.1.1 and 2.1.2), which result in complex probability distributions. To have a convenient way of modelling SCOPs on probabilistic networks, SCOP SOLVER uses SC-ProbLog [29] that extends ProbLog [14, 17] with the possibility of modelling constraints and optimisation criteria. ProbLog is an extension of Prolog such that it allows reasoning over uncertainty and is particularly suited for modelling probabilistic networks [14].

Example 3.1.1. Modelling a viral marketing problem with SC-ProbLog. To illustrate how to model SCOPs using the proposed SC-ProbLog syntax [29], Program 3.1 shows the program that models the viral marketing problem from Example 2.1.1. It first defines the nodes (line 1) and edges (lines 2 and 3) from the network. The edges and their weights correspond to stochastic variables and their probabilities. Then, it defines that the edges are undirected (lines 4 and 5) and for each node a decision variable (line 6) indicating whether we give a free sample to each person. Next, it defines how people can be influenced by their acquaintances and become a customer (lines 7 and 8). Subsequently, it bounds the number of free samples to 2 (line 9). Finally, it defines a probabilistic *query* for all people in our set of interest Φ, in this case person d with `buys(d)`. For all people in Φ, the program maximises the expected number of people buying our product (line 10). SC-ProbLog

Program 3.1: ProbLog program that models the viral marketing problem

```
person(a). person(b). person(c). person(d).
0.4::directed(a,b).     0.8::directed(a,c).
0.1::directed(b,c).     0.3::directed(c,d).
trusts(X,Y) :- directed(X,Y).
trusts(X,Y) :- directed(Y,X).
?::marketed(P) :- person(P).
buys(X) :- marketed(X).
buys(X) :- trusts(X,Y), buys(Y).
{ marketed(P) => 1 :- person(P). } 2.
#maximize{buys(d). => 1 :- person(d). }
```

is able to construct the formula ϕ_d from this program, this is called *grounding* the program [17].

3.2 Compilation stage

The second stage of SCOP SOLVER is the *compilation stage*. In this section we describe this stage, specifically what techniques are used to tackle the complexity of performing WMC as described in Section 2.2.

3.2.1 Knowledge compilation

After we have obtained a weighted propositional formula ϕ that defines the probability distributions for an event in our SCOP, we can evaluate any strategy σ by performing WMC. In Section 2.2 we discuss the complexity of WMC. To tackle this complexity, SCOP-SOLVER uses *knowledge compilation* [12] during the compilation stage. Knowledge compilation is widely used for probabilistic inference [13]. *Compiling* formulas like ϕ to compact datastructures allows for more tractable inference, e.g. calculating $P(\phi \mid \sigma)$. The compilation (and minimisation) of such datastructures is time-consuming, but this is an one-time effort. In the context of SCOP SOLVER we can use the same datastructure to evaluate multiple strategies. Moreover, compact and simple datastructures also allow for simple or efficient design of algorithms that operate on those datastructures.

SCOP SOLVER compiles ϕ to a compact representation in the form of an *Ordered Binary Decision Diagram (OBDD)* [9]. Alternative data structures to OBDDs exist, such as *Sentential Decision Diagrams* [11], but because the solving stage of SCOP-SOLVER is specifically developed for OBDDs and uses properties unique to the structure of OBDDs, we limit this section to the compilation of OBDDs. For an extensive overview on OBDDs, see [4].

Example 3.2.1. Compiling ϕ_d from a viral marketing problem. In Figure 3.1 we show the OBDD representing the formula for ϕ_d in Equation (2.3) from Example 2.1.1. This OBDD represents the probability distributions for a single event in the SCOP, namely the event that person d buys the product. Like in the SCOP, the OBDD consists of the same two Boolean variables: stochastic variables (circular nodes labeled t_{ij})

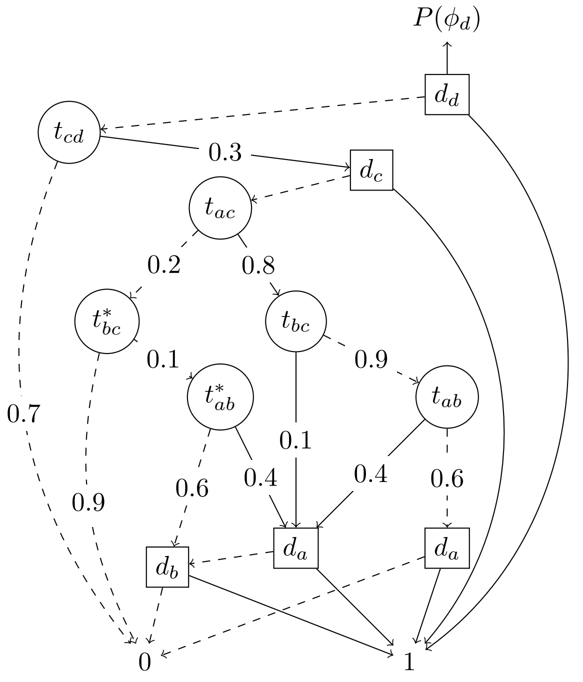


Figure 3.1: OBDD representing ϕ_d from Equation (2.3) with variable order $d_d < t_{cd} < d_c < t_{ac} < t_{bc} < t_{ab} < d_a < d_b$.

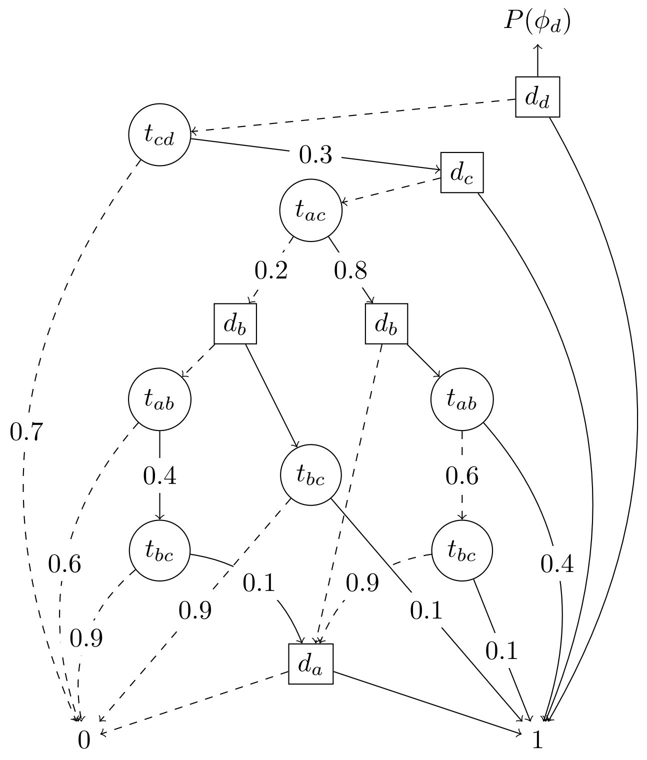


Figure 3.2: OBDD representing ϕ_d from Equation (2.3) with variable order $d_d < t_{cd} < d_c < t_{ac} < d_b < t_{ab} < t_{bc} < d_a$.

and decision variables (squares labeled d_i). Each node in the OBDD has two outgoing edges, the dashed (*lo*) edge corresponds to the case where the variable is *False* and the solid (*hi*) edge where the variable is *True*. The weights on the outgoing edges from the nodes labelled with stochastic variables represent the probabilities of that variable being *True* or *False*. The weights of the outgoing edges of nodes labelled with decision variables are determined by the strategy σ under consideration. For example, if d_d is *True* in σ, its outgoing *hi* edge gets weight 1 and the *lo* edge gets weight 0. The same variable can appear more than once in an OBDD, for example t_{bc} and t_{ab} in Figure 3.1. We use the symbol $*$ to distinguish two nodes labelled with the same variable. Given a combination of truth values to the decision and stochastic variables, we can evaluate ϕ_d by following the outgoing edges for each variable that correspond to these truth values, starting at the root node.

One important property of OBDDs that we want to highlight is the underlying *variable order*. The variable order specifies the order in which variables are encountered during the traversal of the OBDD from root to leaves and has a significant impact on the size and shape of an OBDD. To show how the variable order can affect the OBDD, we show the OBDD for the same formula as in Figure 3.1, but with a different variable order in Figure 3.2. This OBDD is larger and has a different shape.

3.2.2 Weighted model counting with OBDDs

This subsection describes how to perform WMC with an OBDD such that we can efficiently evaluate the probability of $P(\phi \mid \sigma)$ for any strategy σ.

Given an OBDD that represents the probability distributions for ϕ, this OBDD can be mapped to an Arithmetic Circuit (AC) as follows. We assign a *node score*, $v(r)$, for each node r in the OBDD:

$$v(r) = w(r) \cdot v(r^{+}) + (1 - w(r)) \cdot v(r^{-}) \tag{3.1}$$

Each node r is labelled with a variable that has a weight $w(r)$. If this variable is stochastic, $w(r)$ equals the probability that that variable evaluates to *True*. Note that this weight is in that case the same as the label of the outgoing positive edge of the node. For the decision variables, $w(r)$ is 1 if the variable is *True* in the given strategy σ, else it is 0. The node r^+ is the child following the *hi* (solid) edge of r. The node r^- is the child following the *lo* (dashed) edge of r. For the two leaves of the OBDD, 0 and 1, $v(0) = 0$ and $v(1) = 1$ hold.

The probability $P(\phi)$ is equal to the score of the *root* node of the OBDD. We can calculate $v(r)$ (Equation (3.1)) for each node r in the OBDD in a bottom-up traversal to evaluate $P(\phi \mid \sigma)$ for any σ in time linear to the size of the OBDD.

Example 3.2.2. Evaluating $P(\phi_d \mid \sigma)$ with an OBDD. Consider the propositional formula ϕ_d in Equation (2.3) from the viral marketing problem, which is represented by the OBDD in Figure 3.1. Given the strategy $\sigma = (d_a = \top, d_b = d_c = d_d = \bot)$, we calculate $P(\phi_d \mid \sigma)$ by mapping the OBDD to an AC as follows.

First we calculate $v(d_a)$ and $v(d_b)$:

$$v(d_a) = w(d_a) \cdot v(1) + (1 - w(d_a)) \cdot v(0) = 1 \cdot 1 + 0 \cdot 0 = 1$$

$$v(d_b) = 0 \cdot 1 + 1 \cdot 0 = 0$$

Then we can calculate the scores for the two nodes each of t_{ab} and t_{bc}.

$$v(t^*_{ab}) = 0.4 \cdot v(d_a) + 0.6 \cdot v(d_b) = 0.4 \cdot 1 + 0.6 \cdot 0 = 0.4$$

$$v(t_{ab}) = 0.4 \cdot v(d_a) + 0.6 \cdot v(d_a) = 0.4 \cdot 1 + 0.6 \cdot 1 = 1$$

$$v(t^*_{bc}) = 0.1 \cdot v(t_{ab}) + 0.9 \cdot 0 = 0.04$$

$$v(t_{bc}) = 0.1 \cdot v(d_a) + 0.9 \cdot v(t_{ab}) = 0.1 \cdot 1 + 0.9 \cdot 1 = 1$$

Finally, we can calculate $v(t_{ac})$, $v(d_c)$, $v(t_{cd})$ and $v(d_d)$ in that order, to arrive at the root node of the OBDD.

$$v(t_{ac}) = 0.8 \cdot v(t_{bc}) + 0.2 \cdot v(t^*_{bc}) = 0.8 \cdot 1 + 0.2 \cdot 0.04 = 0.808$$

$$v(d_c) = 1 \cdot v(t_{ac}) + 0 \cdot 1 = 1 \cdot 0.808 = 0.808$$

$$v(t_{cd}) = 0.3 \cdot v(d_c) + 0.7 \cdot 0 = 0.3 \cdot 0.808 = 0.2424$$

$$P(\phi_d) = v(d_d) = 1 \cdot v(t_{cd}) + 0 \cdot 1 = 1 \cdot 0.2424 = 0.2424$$

If we sum the probabilities from Table 2.1, which has all the models for the same σ, we get the same value for $P(\phi_d \mid \sigma)$.

$$0.1296 + 0.0144 + 0.0024 + 0.0864 + 0.0096 = 0.2424$$

In case we are interested in other events besides ϕ_d, for example if $\Phi = \{\phi_c, \phi_d\}$, we would also have to evaluate $P(\phi_c \mid \sigma)$ with a different OBDD and sum these probabilities, possibly multiplying with ρ, see Equation (2.1).

3.3 Solving stage

In the *solving stage*, SCOP SOLVER uses the compiled OBDD to find the solution to the problem. In this section we describe how to solve an SCOP once we have obtained the OBDD and thereby an efficient way of evaluating any $P(\phi \mid \sigma)$. A naïve approach for this would be to enumerate every possible strategy σ and evaluate it using the OBDD. Considering that for an SCOP with n decision variables 2^n possible strategies exist, this approach does not scale well. Therefore we need a method that has a more efficient way of traversing the search space of possible strategies. SCOP SOLVER uses techniques from *Constraint Programming (CP)* for this purpose.

3.3.1 Decomposition method

The *decomposition method* [29] converts the OBDD to an Arithmetic Circuit (AC) like described in Section 3.2.2 and decomposes it into a set of linear constraints. For each node in the circuit, a constraint is constructed that represents the score of this node according to Equation (3.1). This decomposition then serves as an input for existing *Mixed Integer Programming (MIP)* solvers or CP solvers. This approach has the disadvantage that during search, there is no guarantee that a partial assignment for the decision variables can lead to a strategy that satisfies the provided constraints and therefore to a feasible solution. For proof see [30].

3.3.2 Solving using constraint programming

To resolve the disadvantage that the decomposition method poses, the solving stage of SCOP SOLVER uses a different approach, also using CP techniques. In this subsection we give an introduction to CP and we describe the solving stage of SCOP SOLVER.

3.3.2.1 Introduction to constraint programming

In Section 2.1.1 we show how we can solve an SCOP by repeatedly solving a constraint satisfaction problem. In practice, a CP solver solves such a problem by iteratively performing two processes, *search* and *propagation*. In the context of the solving stage of SCOP SOLVER this works as follows.

Each decision variable from the SCOP has a *domain* that holds its possible values. Because we consider SCOPs defined on Boolean variables, initially this domain equals $\{0, 1\}$. During the *search* step, the solver selects an *unbound* variable and assigns to this variable a value from its domain. Initially, the set of unbound variables contains all the decision variables from the SCOP.

After each search step, *propagation* updates the domains of the remaining unbound variables. This is done by removing values from the domains of variables that violate the provided constraint(s) of the problem. Because the domains of the variables only hold two values, removing one value from this domain means that the variable becomes *fixed* to the remaining value. If the domain of a variable becomes empty, the (partial) solution found so far is infeasible. By backtracking to the previous search step, we can select a different variable to which we assign a value and arrive at different solutions. For more details on CP, see [2].

3.3.2.2 Search step: branching on decision variables

During the search step an unbound decision variable is selected along with a value that is assigned to this variable. During this step, SCOP SOLVER uses *branching heuristics* to quickly find good partial solutions. A branching heuristic combines a mechanism to select a variable with a mechanism that selects which value to assign first to the selected variable. By backtracking, the solver can possibly assign the other value. Experiments have been performed with different heuristics [31], we describe and discuss these heuristics in Section 5.2.1.

3.3.2.3 Propagation of a constraint on the OBDD

To ensure that we only branch over variables such that the partial solution can lead to feasible solutions, a new method was developed in [30] for SCOP SOLVER that takes care of the propagation for a global constraint on the OBDD.

This constraint is *not* the constraint on the cardinality of the solution (the number of decision variables that are *True* in the corresponding strategy σ). The constraint on the OBDD represents the constraint on the *objective function* from the SCOP, that requires that this value is greater than a threshold θ, see Equation (2.2). This constraint is interpreted as a constraint on the score of the root of the OBDD (see Section 3.2.2 and Equation (3.1)). Because initially all decision variables are unbound, the weights of the outgoing edges of the nodes labelled with those variables (see Section 3.2.2) are still undefined. The constraint on the score of the root of the OBDD to be greater than θ is essentially a constraint on the weights we can put on those outgoing edges.

Given a partial solution (a strategy σ) composed of truth values assigned to decision variables by previous search steps and fixed variables with a domain of size 1. The first approach for propagation of the OBDD constraint is as follows. For every remaining unbound variable d:

1. Create a new strategy σ' that is equal to σ with in addition:
    (a) Variable d temporarily fixed to *False*;
    (b) All remaining unbound variables temporarily fixed to *True*.
2. Calculate $P(\phi \mid \sigma')$: the score of the root of the OBDD.
3. If the score of the root of the OBDD is lower than the current threshold, $P(\phi \mid \sigma') < \theta$, remove the value *False* from the domain of d (d becomes fixed to *True*).

The key to this approach is that the problem under consideration has monotonic probability distributions, see Section 2.1.1. This means that the upper bound of the score of the root of the OBDD is equal to the value it has when we assign *True* to all unbound decision variables. This propagation algorithm has to calculate the score of the root of the OBDD for every unbound decision variable to update their domains. Calculating the score of the root node takes time linear in the size of the OBDD, see Section 3.2.2. Propagation of a constraint on an OBDD of size n, if there are m unbound decision variables, therefore has the complexity of $O(m \cdot n)$.

Because propagation is required after every search step and thus possibly at every node in the search tree, this is inefficient.

3.3.2.4 Propagation using derivatives

To improve over the efficiency of the naïve approach, another propagation algorithm has been proposed that uses the concept of *derivatives* [30]. If we consider the score of the root of the OBDD according to Equation (3.1) to be defined as a *function* with a strategy σ as input, $f(\sigma)$, this propagation algorithm uses the partial derivatives of f with respect to each unbound decision variable. Intuitively, the partial derivative of f with respect to variable d represents the change in the score of the root of the OBDD if we change the value of d from *True* to *False*.

The partial derivative of f with respect to variable d is calculated by:

$$\frac{\partial f(d, \sigma' \setminus d)}{\partial d} = f(\sigma') - f(d = \perp, \sigma' \setminus d) \tag{3.2}$$

where σ' is a strategy that is obtained by taking the partial assignment of decision variables so far, σ, and assigning the value *True* to all the remaining unbound variables from σ. The value of $f(\sigma')$ then represents the upper bound for f given the strategy σ. $f(d = \perp, \sigma' \setminus d)$ is the value of f when we switch the value of d in σ' to *False*.

The propagation algorithm calculates this derivative for all unbound decision variables and checks the following requirement for all:

$$f(\sigma') - \frac{\partial f(d, \sigma' \setminus d)}{\partial d} \geq \theta \tag{3.3}$$

If d does not meet this requirement, this means that if we do not assign *True* to d, we cannot reach the threshold θ with our current strategy σ. Therefore, for all unbound variables d that do not meet this requirement, the value *False* will be removed from its domain. By doing this, the propagation algorithm ensures that the current (partial) strategy σ can lead to a strategy that satisfies the constraint from Equation (2.2).

Because the motivation for using derivatives is to improve on the efficiency of the propagation, an efficient way of calculating these derivatives is key. This is done in an incremental manner in which the derivatives of all unbound decision variables are calculated with a single bottom-up and top-down pass over the OBDD. Instead of evaluating the OBDD for every unbound decision variable, we only need two sweeps to calculate all the derivatives and perform propagation, improving the complexity of one propagation step to $O(m + n)$. We refer to this propagation algorithm as *Full-sweep*. For more details, see [30]. Moreover, in more recent work [31] observations are addressed that further improve the efficiency, resulting in sub-linear propagation of the OBDD constraint. In short, this improvement comes from the idea that it is not necessary to traverse certain parts of the OBDD, whereas the Full-sweep algorithm always traverses the full OBDD twice. Specifically, it is shown that it is only necessary to traverse the part of the OBDD between the borders of the highest and lowest unbound decision variable. This part of the OBDD is referred to as the *active part*. We refer to this propagation algorithm as *Partial-sweep*.

Both the Full- and Partial-sweep propagation algorithms were experimentally evaluated and compared to the method described in Section 3.3.1 and shown their merit in a proof of principle [31]. To be able to apply these algorithms along with the full pipeline of SCOP SOLVER to larger and more complex problems, we believe there is still an effort to be made to optimise this solver. In the following chapters we describe our approach and related work we use for this purpose.

Chapter 4

Programming by Optimisation

To optimise the performance of the solver described in Chapter 3, we apply the *Programming by Optimisation (PbO)* [21] paradigm. In this chapter, we provide an introduction to this concept, and an introduction to *Automated Algorithm Configuration (AAC)*, which enables the PbO paradigm.

4.1 Programming by Optimisation

During algorithm or software development, there are typically multiple ways of performing a single subtask. Each approach could offer different advantages and suffer from other disadvantages possibly depending on its application. However, often only one of these *design choices* is implemented in the final version of an algorithm or software system. This choice is often made based on limited experimentation, with a specific application in mind, while the alternatives are abandoned. These design choices have no effect on correctness, but can affect performance, especially when dealing with computationally challenging problems.

The paradigm of *Programming by Optimisation (PbO)* introduces a different approach to deal with design choices. Developers who take a PbO-based approach to software or algorithm design, implement multiple alternatives for many elements or components. They provide the end user with the choice between these options, by exposing them as configurable parameters. Following the PbO paradigm, developers focus on exploring alternatives for design choices instead of determining the best instantiations for specific applications. Expanding the design space of a given algorithm or software system and exposing choices as configurable parameters provides the basis for using automated algorithm configuration techniques for performance optimisation. The existence of effective automated algorithm configuration procedures hence enables PbO-based algorithm and software design.

4.2 Automated algorithm configuration

When we follow the paradigm of PbO, the resulting algorithm comes with a set of parameters. The parameter settings of such an algorithm (its *configuration*) can have a substantial impact on its performance, and the optimal choice may vary for different sets of problems. This applies to many, if not all, state-of-the-art algorithms that come with multiple parameters. Using the right configuration of such an algorithm is then critical for reaching state-of-the-art performance, especially when applied to NP-hard problems, such as SCOPs.

The process of automatically finding a configuration with optimised performance for a problem set is called *Automated Algorithm Configuration (AAC)* [20]. In this section we introduce AAC by describing the problem it aims to solve. We also give a short overview of some methods that have been developed in order to solve this problem and give some examples of their application.

4.2.1 The algorithm configuration problem

The problem of finding the configuration for which the empirical performance of an algorithm for a given use context is optimised, is called the *algorithm configuration problem* [20]. This problem is defined as follows. Given:

- a target algorithm A;
- a list of parameters $q_1, \ldots, q_n$ of A;
- a configuration space C that defines for each q_ℓ for $\ell \in \{1, \ldots, n\}$ its domain and possible values; each set of such values is a configuration $c \in C$;
- a set of problem instances I;
- a performance metric m that measures the performance of the target algorithm A on the instances of I for a given configuration c,

find a configuration $c^* \in C$ that optimises performance metric m of A on I.

The configuration c^* is expected to be one that performs well on instances similar to the ones in I. Carefully selecting instances for I is therefore important: they need to be representative of the problems on which the optimised algorithm will be applied. If the set of problem instances on which this algorithm was configured, was not representative of new instances, it might not perform well. Another cause for this, could be that the set of instances was too small or contained instances with too much variability between them.

The configuration space C may consist of parameters with different types of domains. *Categorical* parameters have a discrete finite set of possible values and are mostly used to select from a number of possible components of the algorithm. *Integer*- and *real-valued* parameters usually further specify the behaviour of the algorithm and its components. *Boolean* parameters are used in turning off and on settings of the algorithm. Parameters can also be *conditional*. Conditional parameters are parameters that have *dependencies* on other parameters, they are only active if these other parameters are set to particular values.

Different metrics m can be used to measure the performance of the target algorithm A. When using a metric such as median running time, it is possible that runs of A cannot provide solutions in a reasonable time on a large part of the problem instances. This results in poor performance. Therefore, it is common practice to stop these long runs and use a performance metric based on penalised averaging. A fixed penalty is then assigned to runs in which the target algorithm is not able to find a solution within a given time frame.

4.2.2 Different configurators

There are several state-of-the-art methods available that solve the algorithm configuration problem. We refer to these methods as *configurators*. In this section we will describe three of them, *Iterated F-race (I/F-race)*, *Sequential Model-based Algorithm Configuration (SMAC)* and *Gender-based Genetic Algorithm Configuration (GGA++)*.

The configurator *Iterated F-race (I/F-race)* [3] is a configurator that uses *racing procedures*. The concept of racing is as follows. Every possible configuration is evaluated on instances from the set of problem instances and any configurations that perform significantly worse than the leading configuration, are eliminated. This leading configuration is the configuration that has the best performance at that point according to a given performance metric m. Initially, the set of possible configurations can be too large to evaluate entirely. I/F-race was developed such that the search in the configuration space could be more effective and focused on promising candidates, selecting/generating candidate configurations by sampling a probabilistic model. This model consists of probability distributions for the parameters of the algorithm. The configurations that survive each step, are used to update this model, such that the sampling will increasingly favour promising configurations.

Sequential Model-based Algorithm Configuration (SMAC) [23] is a configurator that uses a *model-based* search. In contrast to the probabilistic model used by I/F-race, SMAC builds a model that directly captures the dependency of the performance of the target algorithm on its configuration. This model is used to predict the performance of configurations on multiple instances and to select promising candidate configurations. This method supports diversity of domains of parameters and conditional parameters.

Gender-based Genetic Algorithm Configuration (GGA) is a configurator that uses an approach based on a *Genetic Algorithm (GA)*. *GGA++* [1] is an improvement to GGA and combines a GA with a model-based approach, where in a similar fashion as SMAC, a model is used to predict performance of a configuration. In GGA++, this prediction is used in addition to the crossover operator from the GA to create promising offspring, where each offspring represents a configuration.

4.2.3 Application of AAC

As explained above, AAC is an essential tool for PbO-based software development. It has been applied successfully to algorithms that solve NP-hard problems, such as the *Boolean satisfiability problem (SAT)* [25] and the *Travelling Salesperson Problem (TSP)* [33]. Because solvers designed for problems such as SAT and TSP often have a large configuration space and considering the difficulty of the problems, we also expect the application of AAC on other NP-hard problems, such as SCOPs to be benificial. Moreover, AAC also has been used to configure state-of-the-art MIP solvers to improve their performance [22]. MIP solvers are also highly parametrised and in Section 3.3.1 we briefly mentioned how a MIP solver has been used to solve SCOPs,

Chapter 5

Approach

The goal of the research in this thesis is to optimise the performance of SCOP SOLVER described in Chapter 3. In order to reach our research goal, we follow the PbO paradigm described in Chapter 4. We expose hard-coded design choices in SCOP SOLVER as parameters and add alternatives to these choices as additional parameter values. This enables us to apply AAC (Section 4.2) and automatically optimise the performance of the resulting solver. We identify design choices in the compilation stage and solving stage of SCOP SOLVER. In this chapter we discuss these choices and the alternatives to the current design.

5.1 Configuring the compilation stage

In the compilation stage of SCOP SOLVER, the probability distributions of the problem are compiled to an OBDD that allows for tractable probabilistic inference. In the solving stage this OBDD is used to find the solution to the problem. In Section 3.3 we show that the efficiency of the solving stage of SCOP SOLVER heavily depends on the size of the OBDD. Specifically, the propagation of the global OBDD constraint is linear in the size of the OBDD. The Partial-sweep propagation algorithm could perform even better, if there is a significant difference between the size of the full OBDD and the size of the active part of the OBDD, as described in Section 3.3.2.4. In general, a smaller, more compact OBDD results in improved performance of SCOP SOLVER.

The size of the OBDD, as explained in Section 3.2, can be minimised by finding a good order of the OBDD's variables. Finding the optimal variable order is an NP-hard problem [7] and can therefore be time-consuming. There exist many different variable ordering algorithms, but deciding which algorithm will perform best for which application is not trivial. Therefore the choice between variable ordering algorithms is an interesting design choice. The design space we consider for the compilation stage of SCOP SOLVER consists of: a Boolean variable that indicates whether to minimise an OBDD, or to leave it with default variable order; the different variable ordering algorithms; and their parameters. We summarise this design space in Table 5.1.

In the remaining part of this section, we describe the variable ordering different algorithms we consider, which in short, are the following:

- The *Sifting* algorithm [40];
- Two other variants of the Sifting algorithm, *Symmetric sifting* [37] and *Group sifting* [36];
- Variable ordering using the *Window permutation* approach from [27];
- Variable ordering using *Simulated Annealing* similar to [6] or a *Genetic Algorithm* inspired by [15];
- A random approach for variable ordering, based on the Sifting algorithm.

Table 5.1: SCOP SOLVER parameters relevant to the compilation stage, their domains, short descriptions, and conditions. Except for MINIMISE itself, parameters are conditioned on MINIMISE = *True*.

Compilation stage

Parameter	**Description**
`Minimise`, domain: $\{$*False*, *True*$\}$	Minimise the OBDD.
`VarOrder`, domain: $\{$*Sif*, *SymSif*, *GSif*, *WP*, *SA*, *GA*, *Rand*$\}$	Sifting (*Sif*) [40], Symmetric Sifting (*SymSif*) [37], Group Sifting (*GSif*) [36], Window Permutation (*WP*) [27], Genetic Algorithm (*GA*) [15], Simulated Annealing (*SA*) [6], Random (*Rand*).
`Converging`, domain: $\{$*False*, *True*$\}$	Repeat variable reordering algorithm until no improvement in OBDD size is found (if `VarOrder` $\in$ $\{$*Sif, SymSif, GSif, WP*$\}$).
`MaxSwap`, domain: $\mathbb{N}^+$	Upper bound on number of times two variables can be swapped in the variable order (if `VarOrder` $\in$ $\{$*Sif*, *SymSif*, *GSif*$\}$).
`MaxSift`, domain: $\mathbb{N}^+$	Upper bound on number of variables that are sifted, i.e. moved up and/or down the variable order by swapping with other variables (if `VarOrder` $\in$ $\{$*Sif*, *SymSif*, *GSif*$\}$).
`MaxGrowth`, domain: $\mathbb{R}^+$	Maximum relative increase of OBDD size during minimisation (if `VarOrder` $\in$ $\{$*Sif*, *SymSif*, *GSif*$\}$).
`WSizes`, domain: $\{2, 3, 4\}$	Evaluate different permutations of `WSizes` consecutive variables in the variable order at a time (if `VarOrder` = *WP*).

5.1.1 Sifting algorithms

The *Sifting* algorithm [40] is an algorithm based on the *swap* operation. A swap is an exchange of two adjacent variables in the variable order. The general approach of this algorithm is to move each variable up and down in the order by swapping and saving the best position, this is called *sifting*. With this algorithm there are three extra parameters that can be set. `MaxGrowth` is the maximum increase of size of the OBDD during the sifting of each variable, `MaxSwap` is the maximum number of swaps that can be performed during the sifting of each variable and `MaxSift` is the maximum number of variables that can be sifted. There is also a *converging* version of this algorithm that iterates until there is no longer any improvement to the size of the OBDD, otherwise each variable is only considered once.

The Sifting algorithm also has an *symmetric* variant, which can also be converging or not, called *Symmetric sifting* [37]. Variables that at any point during the variable ordering process become adjacent are tested for *symmetry*, which means that the size of the OBDD is invariant under the swapping of these variables. If this is the case, the variables are then grouped together, which means they are swapped as a group instead of individually.

The *Group sifting* algorithm [36] is a further extension of Symmetric sifting, where the grouping of variables is not restricted just to symmetric variables but can be applied anywhere in the variable order. The general approach in this algorithm is to find a good "neighbourhood" of variables for a single variable and then move the variable and its neighbours around in search of a better position in the variable order. The notion of symmetry is also extended further to better detect variables with strong affinity.

5.1.2 Window permutation approach

The *Window permutation* approach [27] considers only *windows* of w variables at a time, checking each permutation of the variables in that window and keeping the best permutation. The higher the value for w, the more time consuming the algorithm is, but the better the resulting order tends to be. In the implementation we use for our research [42], the available window sizes are 2, 3 and 4, parametrized by `WSizes`. The converging variant repeats itself until there is no longer any improvement found, otherwise the window is shifted over all the variables only once.

5.1.3 Other variable ordering algorithms

Two other approaches that exist, use a *Genetic Algorithm (GA)* [6] or *Simulated Annealing (SA)* [15] to find a good variable order. Currently, the parameters for each respective algorithm are hard-coded in their implementation we are using for our research. In future work, we could further expand the design space of the compilation stage to also include these exposed parameters, e.g., the population size in the case of a GA or the cooling schedule for SA.

The last approach randomly chooses pairs of variables and are swapped in the order. This is done by repeatedly swapping adjacent variables, the best order (in terms of OBDD size) among those obtained by these swaps is retained. The number of pairs chosen for swapping is equal to the number of variables in the OBDD.

5.2 Configuring the solving stage

In Section 3.3.2 we describe how SCOP SOLVER finds a solution to a given problem using CP. In this stage, we consider two design choices. We summarise the design space of this stage in Table 5.2.

The first design choice is which algorithm we use to propagate the constraint on the OBDD, the *Full-sweep* or *Partial-sweep* algorithm. In terms of complexity the Partial-sweep approach has an advantage over the Full-sweep but this does not ensure that it will always result in the best performance in practice [31]. We parametrize this design choice with the `PropVersion` parameter.

The second design choice is which *branching heuristic* to use. As mentioned in Section 3.3.2.2, a CP search algorithm uses branching heuristics to decide in which order unbound decision variables are instantiated and to decide for each variable which value to explore first: *True* or *False*. A good heuristic will be able to prune a large space of the search tree early and traverse the search space faster, thus arrive at the solution faster. Additionally, when solving a constraint satisfaction problem, a good heuristic might also lead to a higher value for the objective function and thus to a higher value for the threshold θ (see Equation (2.2)) which speeds up the search for the optimal value for Equation (2.1). Finding a good heuristic is not trivial and is often a trade-off between time needed for its computation and its potential to make the optimal decisions, therefore we consider this an interesting design choice to explore and optimise.

A branching heuristic consists of a *variable selection heuristic* and a mechanism for *value selection*. The variable selection heuristic determines to which variable we will assign a value and value selection determines whether to assign *False* or *True* to this variable. For simplicity, we combine the selection of the variable and the value into a single notation in this section, but in practice and in Table 5.2 they are separated.

In the remainder of this section we describe the branching heuristics that have been used in earlier work and new alternatives we have implemented. In our description of each heuristic, we describe how the next decision variable is chosen, and indicate which value is assigned to that decision variable first. After backtracking, the CP search algorithm may assign the other value.

Table 5.2: SCOP SOLVER parameters relevant to the solving stage, their domains, short descriptions, and conditions.

Solving stage

Parameter	**Description**
`PropVersion`, domain: {*Linear*, *Sub-linear*}	Version of the propagation algorithm.
`SelectionHeur`, domain: {*Top*, *Bottom*, *Derivative*, *Influence*, *Betweenness*, *Triangle*, *Similarity*, *Simmelian*, *Degree*, *Forest-fire*, *Random*}	Heuristics used for selection of variables.
`ValueHeur`, domain: $\{0, 1\}$	Heuristic that determines whether to select the variable with the highest value for `SelectionHeur` and assign *True* or the lowest value and assign *False*.
`TimeSteps`, domain: $\mathbb{N}^+$	Maximum number of timesteps in which nodes can be influenced (if `SelectionHeur` = Influence).
`NodeSamples`, domain: $\mathbb{N}^+$	Number of node samples used to estimate betweenness (if `SelectionHeur` = *Betweenness*).
`FireProbability`, domain: $\mathbb{R} \in [0, 1]$	Probability that the fire spreads from one node to a neighbour (if `SelectionHeur` = *Forest-fire*).
`EdgesBurnt`, domain: $\mathbb{R} \in [0, 1]$	Minimum fraction of the total number of edges in the graph that have to burned (if `SelectionHeur` = *Forest-fire*).

5.2.1 Existing heuristics

As mentioned in Section 3.3.2.2, Latour *et al.* [31] performed experiments with six different branching heuristics. These heuristics use information from the OBDD compiled in the compilation stage.

Top-0, *Top*-1, *Bottom*-1 and *Bottom*-1 are static heuristics and are inspired by the size and shape of the OBDD. *Top*-0 (*Bottom*-0) branches on the highest (lowest) unbound decision variable in the OBDD's variable order and assigns to this variable the value *False* first. Their counterparts *Top*-1 (*Bottom*-1) assign the value *True* first. In Section 3.3.2.4 we describe how the Partial-sweep propagation algorithm only has to traverse the active part of the OBDD. The intuition behind these four heuristics is that by repeatedly selecting decision variables that lie on the borders of the active part of the OBDD, we iteratively decrease the computation time of this propagation algorithm.

The fifth and sixth heuristic use the calculated derivatives during the propagation of the constraint on the OBDD. As explained in Section 3.3.2.4, the derivative of variable d represents the change in the value on the root of the OBDD (Equation (3.1)) if we change the value of d from *True* to *False*. Consequently, a variable with a high derivative has a large impact on the objective function. The heuristic *Derivative*-0 (*Derivative*-1) assigns the value *False* (*True*) to the variable with the *smallest* (*largest*) absolute derivative. The derivatives of variables change during the solving stage and are dynamic, recalculated after each propagation.

5.2.2 New heuristics

We propose new heuristics that take a different approach: they are derived directly from the network/graph on which the SCOP under consideration is defined. For this purpose, we explored relevant work from the network science literature.

In Examples 2.1.1 and 2.1.2 we describe two examples of an SCOP. An important difference between the two is what the decision variables in the problems represent. In the viral marketing problem, decision variables represent the choice whether to hand out a free sample to a person, represented by nodes in our social network. Each node has a corresponding decision variable. However, in the powergrid reliability problem, the decision variables represent the choice whether to reinforce a powerline, represented by edges, in the powergrid. Thus, each edge has a corresponding decision variable. This means that the heuristics that we define based on the graph of a problem need to be used to select both edges and nodes.

5.2.2.1 Social influence

The first set of new heuristics we propose use an approximation of the *influence* that a node has on the other nodes in the network. This approach is inspired by work on social influence [8] that focuses on the *influence maximisation problem* for networks. This problem has similarities to the viral marketing problem (Section 2.1.2) in the sense that it aims to find the set of most influencial people in a network. In this problem the *independent cascade model* for spread of influence [28] is used. In this model, nodes are able to spread influence to their neighbours, where each neighbour can then in turn influence their own neighbourhood. This spread of influence depends on the probability represented by the weight on the edges between nodes. The same model could for example be applied to the network in Figure 2.1.

We calculate the influence of a node u as follows. Starting at *timestep* 0, node u is seeded with influence. Node u can then spread this influence to its neighbours. For each neighbour v of u, u can influence v according to the probability that is the weight of edge (u, v). If neighbour v is *influenced*, it can then in turn influence its own neighbours. At each timestep t, all influenced nodes can influence their neighbours. We limit the number of timesteps by $T \in \mathbb{N}^+$. The influence of node u is the number of unique nodes that is influenced at the end of timestep T. The number of timesteps in which nodes can influence each other by a maximum of `TimeSteps` (a configurable parameter of this heuristic). *Influence*-0 (*Influence*-1) first assigns the value *False* (*true*) to the variable with the *smallest* (*largest*) influence.

To use this heuristic to select decision variables associated with edges from a network, we do the following: the influence of an edge (u, v) is equal to the sum of the influences of nodes u and v. We think this is a good measure in the sense that it represents the importance of the nodes an edge connects. Moreover, in the context of a powergrid reliability problem, we do not include any probabilities in the calculation of the influence of a node. Consequently we simply sum the number of unique nodes in the the `TimeSteps`-neighbourhoods of the endpoints of each line. This is because, in our dataset (see Section 6.1.1), $p_{l,1}$ is the same for each line l, and the value of the probability on each line of the network depends on the value of the associated decision variable. Therefore, if we do not take a strategy into account, the probabilities are all equal and it does not serve a purpose to include them in the calculation of this heuristic. Even though we lose the stochastic component of the heuristic in this case, it will still be able to distinguish edges that can lead to a high number of unique nodes, which can be used as a measure of importance for the distribution of power.

5.2.2.2 Betweenness centrality

In network science, centrality measures are often used to identify the most important nodes in a graph. A popular centrality measure to identify both important edges and important nodes is the *betweenness centrality* [19]. The betweenness centrality is the sum of the fraction of all-pairs shortest paths that pass through either a given node or edge. Nodes or edges through that lie on many shortest paths are important for the flow of information through the network. For example, in the context of a viral marketing problem, the information about an advertised product, or the distribution of power in the context of a powergrid reliability problem. Therefore we propose to use this measure as a branching heuristic. The exact value for this heuristic can also be approximated by taking a sample of nodes and only using those to estimate the betweenness, the size of this sample is parametrized with `NodeSamples`. *Betweenness*-0 (*Betweenness*-1) first assigns the value *False* (*true*) to the variable with the *smallest* (*largest*) betweenness centrality.

5.2.2.3 Graph sparsification methods

In the powergrid reliability problem, we assume that powerlines fail to function during a natural disaster and our goal is to preserve those powerlines that are essential to connect consumers of power to producers. Essentially, this has similarities to the nature of graph sparsification, that reduces the size of the network while preserving certain properties. In recent work on graph sparsification [32], methods are described for edge sparsification that can be understood as methods for rating edges by importance and then filtering globally by these scores. We propose to use these edge scores as a heuristic for selecting suitable branching candidates. Implementation is provided of the scoring methods in the `NetworKit`[1] toolkit and assumes the graphs are unweighted.

Because the scoring methods do not take the edge weights into account, in the context of the viral marketing problem, the proposed heuristics assume that all stochastic influence relationship have an equal probability. The loss of this stochastic component means that the heuristics would not favour nodes over others if it is only distinguished by influence relationships with a high probability.

The first method uses the number of *triangles* an edge is a part of in the graph. The number of triangles is often used in the calculation of graph statistics such as the global and local clustering coefficient. We assume the graph is undirected when calculating the number of triangles, such that the requirement for a triplet to count as a triangle remains simple. *Triangle*-0 (*Triangle*-1) first assigns the value *False* (*true*) to the variable that has the *lowest* (*highest*) triangle. We can calculate this for both decision variables that are defined on edges or nodes.

Two other proposed methods aim to sparsify the graph in order to improve the speed and quality of community detection algorithms. Specifically, these methods try to achieve local instead of global sparsification [41]. The *local similarity* score is based on the similarity between the neighbourhoods of two given nodes. With this method, edges are ranked according to the similarity of the nodes it connects and it avoids to destroy structures within local communities. The *quadrilateral simmelian backbone* score uses the number of quadrangles containing an edge to rank them. The argument for this approach is that it is capable of discriminating edges that are placed within and between dense subgraphs and able to identify strong ties that hold together communities [34]. We propose to use these edges scores as a branching heuristic to identify important edges within local communities. This is in contrast to the heuristics that use betweenness centrality, which could aim to identify edges that serve as a bridge between different communities. *Similarity*-0 (*Similarity*-1) first assigns the value *False* (*true*) to the variable that has the *lowest* (*highest*) local similarity score. *Quadrangle*-0 (*Quadrangle*-1) first assigns the value *False* (*true*) to the variable that has the *lowest* (*highest*) score according the quadrilateral simmelian backbone method. To use these edges scores to identify important nodes in the same manner, we rank nodes based on the scores of all its outgoing edges. To our knowledge we have three sensible options for this, taking the sum, the average or the maximum of these edges scores. We use the sum of the edges scores. If we would use the average or the maximum, we would rank nodes with a single important edges over a node with a large number of perhaps insignificantly less important edges, which we want to avoid.

The *edge forest fire* method follows the idea that nodes are burned during a fire that starts at a random node and spreads from neighbour to neighbour. Instead of a standard random walk, this fire can spread to more than one neighbour at the same time (similar to the previously described *influence*) but already burned neighbours cannot be burned again. The score calculated using this approach is based on the idea that edges that are visited more frequently during these walks are more important. *Forest-fire*-0 (*Forest-fire*-1) first assigns the value *False* (*true*) to the variable that has the *lowest* (*highest*) score according the edge forest fire method. In the same

[1]Available at `https://networkit.github.io/`.

fashion as previously described, we use the sum of the scores for all (outgoing) edges a node to calculate its value. Additionally, this heuristic has parameters to set the (constant) probability that a fire spreads from one node to its neighbour, `FireProbability` and a minimum fraction of the total edges in the graph that have to burned, `EdgesBurnt`.

The final method is called the *local degree* of an edge. With this method, the goal is to keep those edges in the sparsified graph that lead to nodes with a high degree. This is based on the assumption that nodes with a high degree serve as a "hub" that is crucial for a complex network's topology and edges that lead to those hubs, serve as a "hub backbone". *Degree*-0 (*Degree*-1) first assigns the value *False* (*true*) to the variable that has the *lowest* (*highest*) local degree score. For decision variables associated with a node, we simply use the degree of each node.

5.2.2.4 Random

We also include a *Random* heuristic that should serve as a baseline and always selects a random decision variable. For consistency in our implementation, this heuristic also generates values for each variable and selects the one with the highest/lowest value and assigns to this a random value. For this purpose it generates an uniform random number $X \in [0, 1]$ for each decision variable. *Random*-0 (*Random*-1) first assigns the value *False* (*true*) to the variable with the highest value for X. This heuristic should serve as a sensible baseline.

Chapter 6

Experiments

To evaluate our approach described in the previous chapter we automatically configured SCOP SOLVER for different applications and evaluated the configured solver. In this chapter we describe the setup for these experiments, the results and their analysis.

Our experiments are guided by the hypothesis that exposing parameters of SCOP SOLVER, providing alternative design choices and automatically configuring the resulting algorithm for any set of given SCOP instances, provides a configured SCOP SOLVER that outperforms the original in terms of running time and number of solved instances for a given cutoff time of ten minutes.

6.1 Experimental setup

In this section we describe the setup for our experiments: a description of the datasets, the experimental protocol, the hardware and the software.

6.1.1 Datasets

Our hypothesis says that with our approach we can improve the performance of SCOP SOLVER for any set of given SCOP instances. To test this, we performed experiments on two different datasets. We summarise some characteristics of these datasets in Table 6.1.

6.1.1.1 Viral marketing dataset

We formulated a viral marketing problem on directed multi-graph data from Facebook representing user interactions [43]. This dataset consists of 46 952 nodes (users) and 876 993 edges (wall posts). We used community detection [5] to extract all communities of twenty to thirty nodes. To convert these communities into probabilistic networks, we used the *independent cascade model* for spread of influence [28]. When a user posts multiple messages on the wall of another user, there are multiple (parallel) edges between these two users. Parallel edges from node u to v are replaced by a single edge with weight of $1 - (1 - p)^{c_{uv}}$, where c_{uv} is the number of edges

Table 6.1: Size of the training and test of both the viral marketing dataset and powergrid dataset and the size of the individual problem instances.

Dataset	Size training	Size test	Nodes	Edges
Viral marketing	197	196	20–30	28–96
Powergrid	72	66	20–70	22–70

from u to v, and $p = 0.1$ is a constant probability, which is interpreted as the probability a single wall post has to influence the user that receives it.

The *set of interest* Φ on which we define our objective function consists of the fifty percent highest-degree nodes in a community. We chose an upper bound of $k = 10$ on the cardinality of the solution for all networks. The size of Φ and value of k are the result of initial experiments to achieve a reasonable running time (see Section 6.1.2).

The resulting set contains 393 problem instances, which we divided into a training and test set such that their distributions of communities with different numbers of nodes are the same. The training set conists of 197 instances and the test set of 196 instances.

6.1.1.2 Powergrid dataset

The instances of the powergrid reliability problem are defined on network models of the European and North-American high-voltage power grids [44], extracted by `GridKit`[1]. These networks are undirected graphs consisting of power producers, consumers and minor grid nodes (nodes) and powerlines that connect them (edges).

For each powergrid from a single European country or North-American state, we extracted the greatest connected components that contain at least one power producer. We selected the components that have at least twenty, and at most 70 nodes and 70 edges, resulting in a set of 23 networks. The set of interest Φ, consists of a randomly selected set of power consumers for each network. We set the size of Φ equal to 50% of the total number of power consumers in each network, similarly to the instances from the viral marketing dataset.

We used a probability of 0.4 that a powerline remains intact during a natural disaster and 0.875 if it is reinforced [16]. We assume a uniform cost of $\gamma_l = 1$ for reinforcing a powerline l. We chose a budget of $\beta = 10$ for all problem instances, such that the constraint on the cardinality of the solution is similar to that of the instances in the viral marketing dataset.

Of the 23 networks, we distribute 12 (randomly selected) networks to the training set and 11 to the test set. For each network, we create six instances with a different Φ such that our dataset consists of a reasonable number of instances. The training set contains 72 instances and the test set 66 instances.

[1]Available from `github.com/bdw/GridKit`.

6.1.2 Protocol

For our experiments we used SMAC [23] as our configurator, because it is a high-performing general purpose configurator and freely available. Moreover, because of its model-based approach it is able to handle parameters with different types of domains, which is important due to the parameters of our solver (see Table 5.1 and Table 5.2).

For each of the datasets described in Section 6.1.1, we performed fifteen independent 48-hour runs of SMAC on the training set. We minimised PAR10 (penalised average running time with penalty factor 10) and a cutoff time of 600 CPU seconds, meaning that we measured the average running time of SCOP SOLVER over all instances in a given set and counted each timed-out run as ten times the cutoff time.

The cutoff time was selected based on initial experiments on the viral marketing dataset, such that a reasonable success rate was achieved for the default configuration. The time limit for each SMAC run was adjusted according to the cutoff time, following the example of scenarios with a similar cutoff time from AClib, a well-known configuration library [26].

For each of these fifteen runs, we evaluated the final incumbent (the configuration with the best PAR10 value) on the training set and selected the configuration with the best performance. We then evaluated this *optimised configuration* on the test set and compared it with the *default configuration*. This default configuration of SCOP SOLVER was based on the results from previous experiments performed by Latour *et al.* [31], thus providing a strong baseline for our configuration experiments. The default configuration does not perform any OBDD minimisation, uses the Partial-sweep propagation algorithm and *Derivative*-1 as a branching heuristic.

6.1.3 Hardware and software

SCOP SOLVER makes use of an SC-ProbLog version based on `ProbLog 2.1` [17] for modelling, the `dd 0.5.4` library [18] for OBDD compilation and the `Scala 2.12` library `OscaR 4.0.0` [35] for solving. We used the Cython binding of the `dd` library to `CUDD 3.0.0` [42] for the implementation of alternative minimisation methods for the OBDDs. The heuristics based on the underlying graphs of problem instances are calculated using `NetworkX 2.2` and `NetworKit 5.0.1`. Our configuration experiments were run on a cluster with 32 nodes, each equipped with 94 GB RAM and two Intel Xeon E5-2683 CPUs with 16 cores, running at 3.0 GHz using CentOS Linux 7.6.1810; for our configuration experiments, we used SMAC [23] v3. The memory limit SCOP SOLVER is allowed for a single run was set to 15 GB RAM.

6.2 Results

To test our hypothesis on both of our datasets we evaluated the performance of the optimised configuration and that of the default configuration on both the training and test set. In this section we present and discuss these results.

6.2.1 Viral marketing dataset results

The optimised configuration for the dataset of viral marketing problems uses the Symmetric sifting algorithm for variable ordering in the OBDD along with specific values for its parameters (`MaxSwaps`, `MaxVars`, `MaxGrowth`). In the solving stage it uses the Full-sweep propagation algorithm and *Betweenness*-1 as branching heuristic.

The PAR10 values for both configurations are shown in Table 6.2 and Figure 6.1a. On the training set, the optimised configuration yields a 74% decrease in PAR10 value compared to the default configuration and on the test set a 88% decrease. Table 6.2 also shows the PAR10 values for those instances that were solved by either configuration, or both. These values represent the performance on the solvable instances within this cutoff time. The increase in performance on these instances, when we compare the optimised configuration to the default, is even higher than that on all instances (a decrease in PAR10 of 98% on both the training and test set).

The running times of both configurations on individual instances are shown in Figure 6.2a. In this figure each datapoint represents a single instance and we can see how much faster the optimised configuration is able to solve it compared to the default configuration. We also indicate the total number of variables (*stochastic* and *decision*) of each instance, which in this case is a good indication of how long it takes to solve an instance.

As the running time of the default configuration increases, the speedup (how much faster a solution is found when comparing configurations) also tends to increase. The optimised configuration is able to achieve a maximum speedup (on an instance solved by both configurations) of a factor 51 on the training set and of a factor 45 on the test set. The optimised configuration is also able to find a solution for many more instances than the default configuration, while there is not a single instances that is solved by the default but not by the optimised configuration. As shown in Table 6.3, the optimised configuration is able to solve 28 out of 197 more instances on the training set and 26 out of 196 more on the test set. This is an increase in solved instances (on both training and test) of 17%.

6.2.2 Powergrid dataset results

The optimised configuration for the dataset of powergrid reliability problem uses the Simulated annealing algorithm for OBDD minimisation. In the solving stage it uses the Full-sweep propagation algorithm and *Derivative*-1 as branching heuristic.

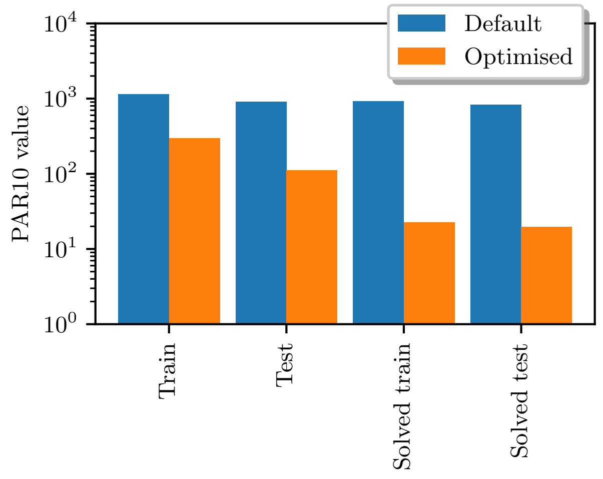
From the PAR10 values in Table 6.2 and Figure 6.1a we can observe a decrease of 34% in PAR10 value if we compare the optimised configuration to the default on the training set and a decrease of 12% on the test set. If we only consider the solved instances, the increase in performance that the optimised configuration achieves is much higher with a decrease in PAR10 of 96% on the training set and 95% on the test set.

For this dataset, the optimised configuration is able to achieve a maximum speedup of a factor 13 on the training set and of a factor 17 on the test set. While it is clear that the optimised configuration generally outperforms the default configuration, there are a few instances on which the default configuration is able to find a solution faster than the optimised configuration. We discuss this in Section 6.2.3.

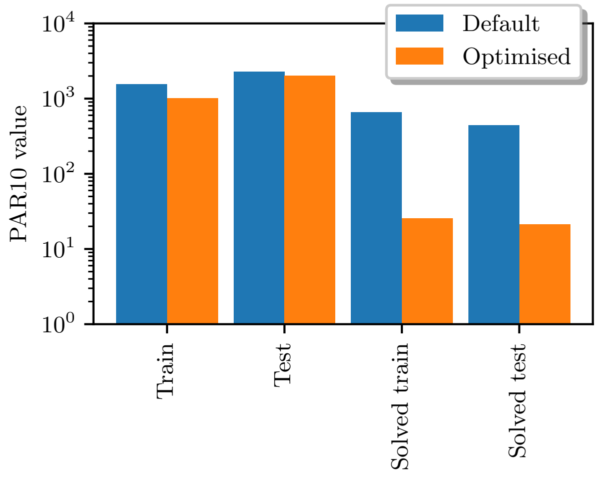
If we look at the number of timeouts in Table 6.3, we can see that the optimised configuration is able to solve

Table 6.2: PAR10 values with a cutoff of 600 CPU seconds of both configurations on the instances from the viral marketing dataset and powergrid dataset. Values are calculated for all instances and only the instances that were solved by either configuration within the cutoff time.

		All instances		Solved instances	
Dataset	Configuration	Training	Test	Training	Test
Viral marketing	Default	1 151.4	914.1	919.3	835.0
	Optimised	295.6	111.2	22.5	19.7
Powergrid	Default	1 546.8	2 294.9	656.2	442.3
	Optimised	1 021.5	2 014.2	25.8	21.2



(a) Viral marketing dataset.



(b) Powergrid dataset.

Figure 6.1: PAR10 values presented in Table 6.2 visualized as bar plot.

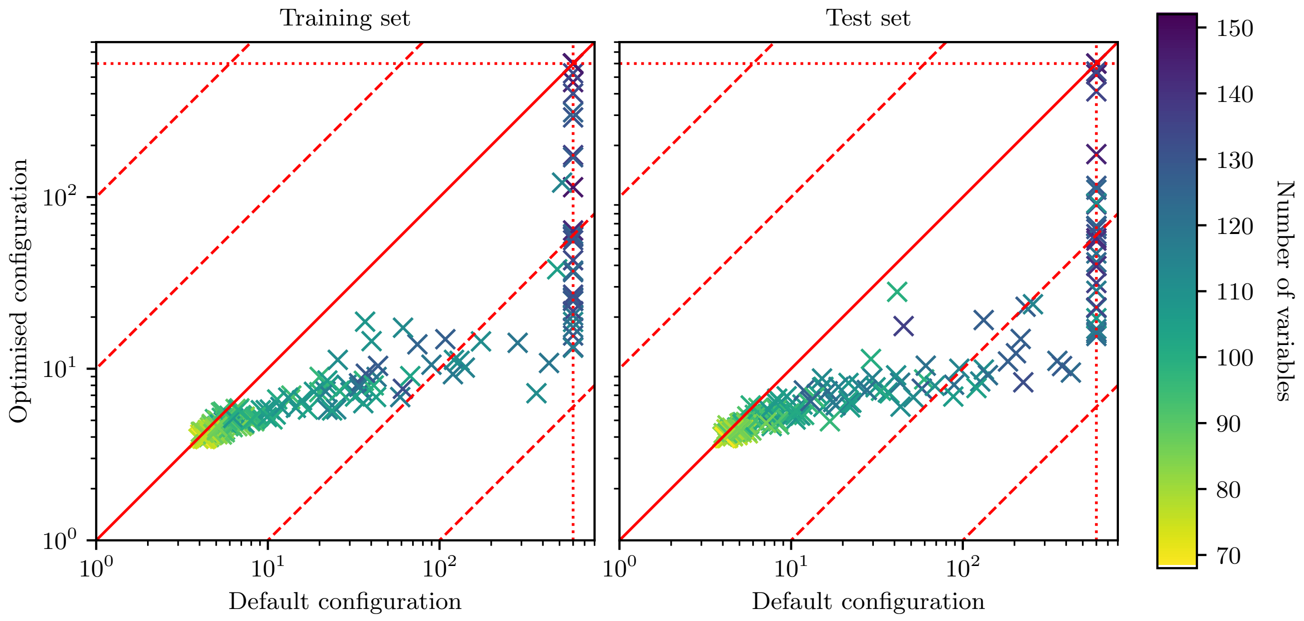
more instances than the default. Specifically, 5 out of 72 more instances from the training set and 3 out of 66 more on the test set, an increase in solved instances of 9%.

6.2.3 Conclusion and discussions

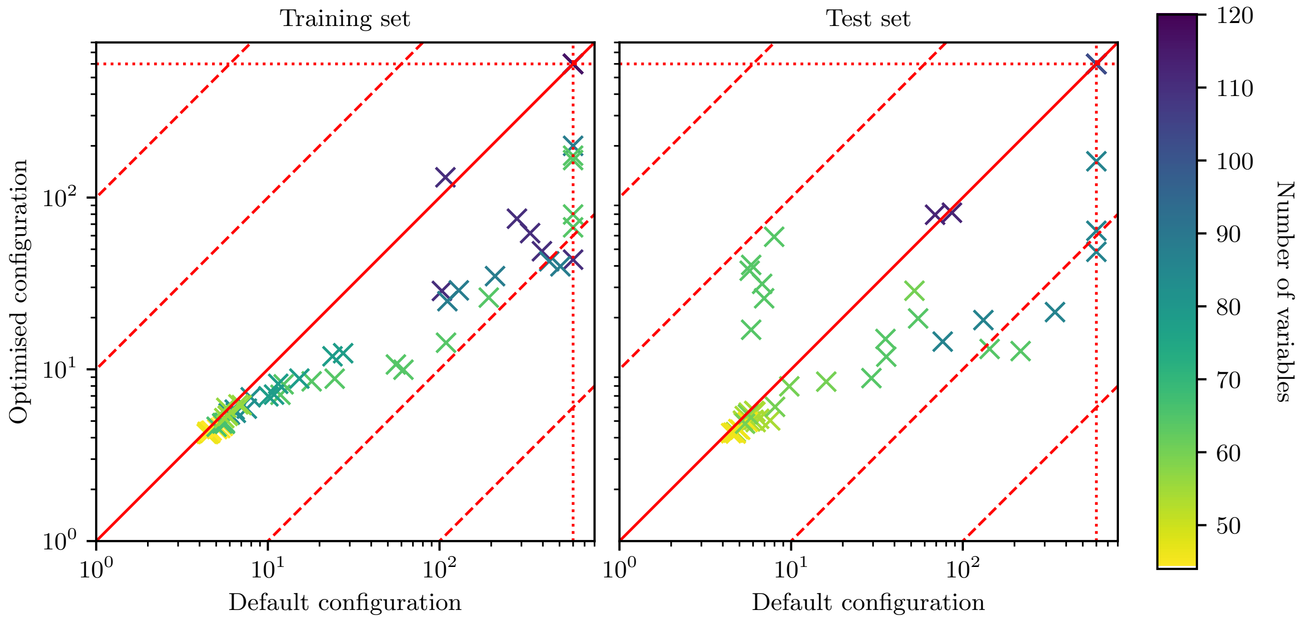
In this subsection we draw our conclusions from the presented results. We also further analyse and discuss any results that warrant this.

6.2.3.1 General conclusion

In terms for running time on individual instances and in PAR10 value, on both datasets, the optimised configuration outperforms the default configuration. This confirms our hypothesis for these sets of SCOPs. We attribute this to the fact that that by applying AAC, we found a previously unexplored configuration of our highly parametric SCOP SOLVER framework that is better suited to the problems from these datasets than the default configuration.



(a) Viral marketing dataset. The default configuration has 37 timeouts on the training and 29 on the test set. The optimised configuration has 9 timeouts on the training and 3 on the test set.



(b) Powergrid dataset. The default configuration has 17 timeouts on the training and 25 on the test set. The optimised configuration has 12 timeouts on the training and 22 on the test set.

Figure 6.2: Running time [CPU s] of both configurations on both datasets, with a cutoff time of 600 seconds (indicated by horizontal and vertical dotted lines). The diagonal dashed lines represent differences in running time of factors of 10 and 100.

Table 6.3: Number of timeouts of both configurations on the instances from the viral marketing dataset and powergrid dataset. The total number of instances for the viral marketing dataset are 197 and 196 for training and test respectively, for the powergrid dataset they are 72 and 66 (see Table 6.1).

Dataset	Configuration	Training	Test
Viral marketing	Default	37	29
	Optimised	9	3
Powergrid	Default	17	25
	Optimised	12	22

6.2.3.2 Comparing configurations

Following from this conclusion we can also take a closer look at the the differences in the optimised configurations for both datasets and the default. For both datasets, performing OBDD minimisation by improving the variable order results in better performance overall. As discussed in Section 5.1, a minimised OBDD can attribute to improved performance because it saves computation time in the solving stage. Between the two datasets there is a difference in which variable order algorithm results in the best performance on the training set. This shows that for different applications, different algorithms can achieve the best results in terms of size and shape of the minimised OBDD and computation time required to find the corresponding variable order. The second parameter for which both optimised configurations differ from the default is the version of the propagation algorithm, where the Full-sweep algorithm apparently offers better performance on the two datasets. We suspect that this is due to the computation required for updating the data used to know which parts of the OBDD are unnecessary to traverse during propagation. The computation time required for this could outweigh the time it saves with more efficient propagation. Lastly, the optimised configuration for the set of viral marketing problems also uses a different branching heuristic. For the viral marketing problems, using *Betweenness*-1 as branching heuristic is better than using the derivatives. Possibly because for social networks (on which we define these problems), the betweenness centrality is an especially good indication of importance from the perspective of control on the flow of information. This corresponds well to the viral marketing problem setting, because the goal is to spread information about an advertised product in this setting.

6.2.3.3 Comparing results between datasets

An observation that warrants discussion is that while we have concluded that the optimised configuration outperforms the default configuration, the improvement we achieve over the default configuration on the powergrid dataset is less pronounced than that achieved for the viral marketing dataset. This is especially true for the PAR10 value over all instances from the dataset. We attribute this to the high number of timeouts that occur in this dataset, relative to the number of timeouts on the viral marketing dataset. From all instances of the training set 17% is currently not solved within the cutoff time by either configuration and 33% of the instances in the test set. These instances have a very high influence on the PAR10 values. As previously noted, the increase in performance on the solved instances is much higher than on all instances. When we increase the cutoff time from

600 seconds to 2 400 seconds and evaluate both configurations on the test set again, the PAR10 value default configuration is 9 111.6 and that of the optimised configuration 5 978.8. This is a decrease in PAR10 of 34% and the optimised configuration has 16 timeouts while the default configuration has 25. The increase in cutoff time results in a significant increase in solved instances while the default configuration does not even slightly profit from this. See Section A.2 for all results of the evaluation with the increased cutoff time.

6.2.3.4 Outliers in the powergrid dataset

In Figure 6.2b there is one group of instances that stands out on which the default configuration significantly outperforms the optimised configuration. These six instances are all derived from the powergrid network from the Mexican state Chihuahua. These instances are not necessarily difficult to solve, considering the default configuration finds a solution within ten seconds. However, minimising the compiled OBDD for these instances has a significant negative effect on the total running time. If we run the same configuration without any OBDD minimisation these instances are also solved within ten seconds.

Chapter 7

Conclusion

We presented an approach to automatically optimise the configuration of a high-performance method for solving *Stochastic Constraint Optimisation Problems (SCOPs)* [31]. Following the *Programming by Optimisation* paradigm [21], we considered alternatives to the design choices made for several key components of this method and exposed those as parameters. We then applied *Automated Algorithm Configuration* to the resulting, highly parametric algorithm framework, using SMAC [23], in order to optimise its configuration for a set of SCOPs. We evaluated the running time of the automatically configured solver against that of expert-chosen default settings, the one that was the best performing configuration from the experiments of Latour *et al.* [31], on two benchmarks.

On a set of viral marketing problems, the optimised configuration solved 17% more instances than the default configuration within a cutoff time of ten minutes, and achieved up to a 51-fold speedup on the solved instances. For a set of powergrid reliability problems, the optimised configuration solved 9% more instances within the same cutoff time and achieved up to a 17-fold speedup on the solved instances.

In future work, we will perform experiments on another dataset, which is composed of instances from a third problem setting. This problem setting is a variant of influence spreading applied to citation networks that extends the viral marketing problem with an extra requirement resulting in an itemset enumeration problem. Other research directions could include the expansion of the design space for different parts of the solver, considering a large part of the computation time is spend on grounding the program of the instances and compiling the *Ordered Binary Decision Diagram* in the modelling and compilation stage of the solver. To decrease the time spent during these steps is essential if we want to solve instances that are currently unsolvable with this solver.

Appendix A

Additional Results

In addition to the results presented in Section 6.2, we present results from some additional experiments in this appendix.

A.1 Parameter importance

Using the data from our configuration experiments we can analyse the importance of the parameters of SCOP SOLVER to determine which parameters have the most influence over the algorithms performance. For this purpose we use `PyImp`, a parameter importance analysis tool and its implementation of the *functional ANOVA (fANOVA)* algorithm [24] to quantify the importance of parameters.

In Table A.1 and Table A.2 we show the five most important parameters according to the fANOVA analysis for the two datasets. The values calculated and shown in the third column of the table are percentages that show how much variance across the whole configuration space, during configuration, can be explained by that parameter. Intuitively, if a parameter is responsible for a higher variance, this means that this parameter can have a larger effect on the performance of SCOP SOLVER.

For both datasets, the parameters that determines which algorithm is used for ordering the OBDD's variables is deemed the most important. The two parameters that determine the branching heuristic, `SelectionHeur` and `ValueHeur` are also ranked highly. Therefore we can conclude that there is a significant difference in performance between the variable order algorithms and branching heuristics we consider in the design space of SCOP SOLVER.

Surprisingly, the parameter that determines whether or not to perform OBDD minimisation, (`Minimise`), is not high ranked. This can be explained by the fact that there is a lot of variance between the minimisation methods, or variable order algorithms, as seen by the high ranking of the `VarOrder` parameter.

Table A.1: Parameter importance determined by fANOVA on the viral marketing dataset.

Rank	Parameter	Variance (%)
1	`VarOrder`	40.0
2	`SelectionHeur`	4.2
3	`PropVersion`	2.0
4	`ValueHeur`	1.2
5	`MaxGrowth`	0.6

Table A.2: Parameter importance determined by fANOVA on the powergrid dataset.

Rank	Parameter	Variance (%)
1	`VarOrder`	25.0
2	`SelectionHeur`	9.2
3	`MaxGrowth`	7.9
4	`ValueHeur`	2.0
5	`PropVersion`	0.7

A.2 Additional results powergrid dataset

In Section 6.2 we mention that we also performed evaluation experiments on the powergrid dataset with an increased cutoff time, because of the high number of timeouts on the test set. In this section we present results from an evaluation of the same default and optimised configuration on the test set, but with a cutoff time of 2 400 seconds instead of 600 seconds. In Table A.3 we can see that this results in a decrease of PAR10 of 34% if we compare the optimised configuration to the default configuration. In Figure A.1 we can see the running times on the individual instances and the number of timeouts. The optimised configuration is able to solve 22% more instances than the default.

Table A.3: PAR10 values with a cutoff of 2400 seconds of both configurations on the instances from the powergrid dataset on all instances and only the instaces that were solved by either configuration within the cutoff time.

Configuration	All instances	Solved instances
Default	9 111.6	4 347.3
Optimised	5 978.8	212.0

A.3 Comparison to existing MIP solvers

In Section 3.3.1 we describe the decomposition method [29] that converts the OBDD compiled in the compilation stage of SCOP SOLVER to an Arithmetic Circuit and decomposes this into a set of linear constraints. This decomposition can then serve as an input for *Mixed Integer Programming (MIP)* solvers. We performed configuration experiments for such a MIP solver on the MIPs created from the decomposition of the OBDDs

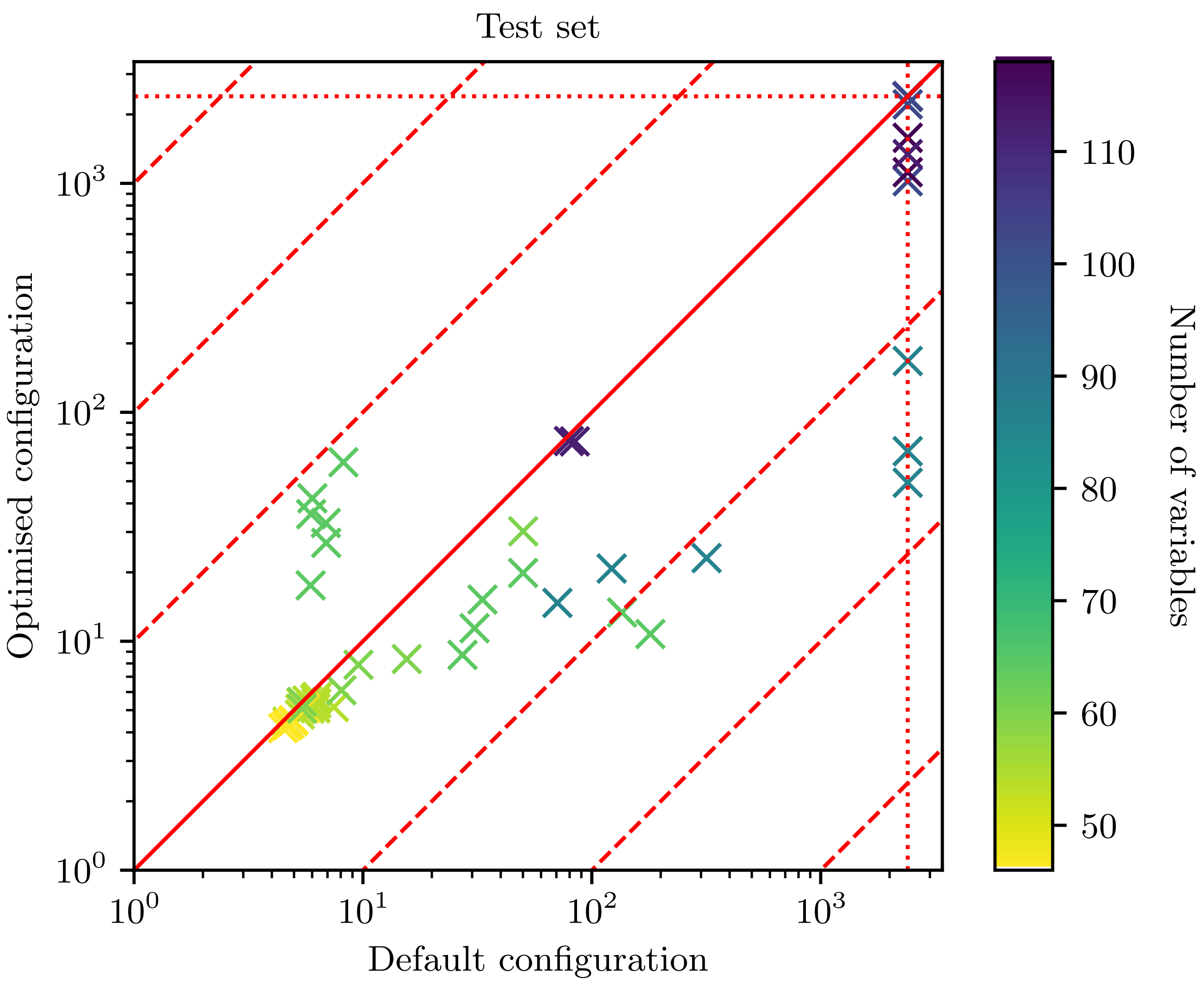


Figure A.1: Running time [CPU s] of both configurations on the test instances from the powergrid dataset, with a cutoff time of 2 400 seconds (indicated by horizontal and vertical dashed lines). The diagonal dashed lines represent differences in running time of factors of 10, 100 and 1 000. The default configuration has 25 timeouts. The optimised configuration has 16.

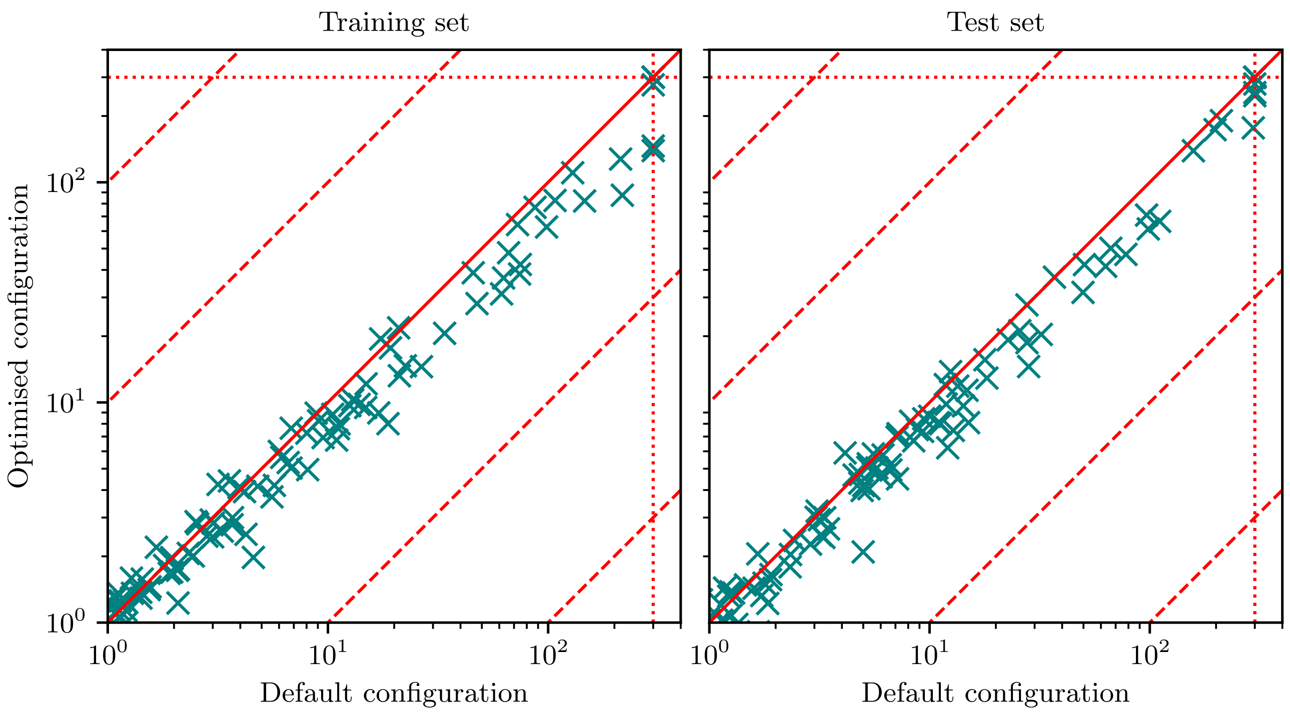
compiled by the optimised configuration of SCOP SOLVER on both our dataset of viral marketing problems. We compare the performance of this solver to the optimised configuration of the solving stage of SCOP SOLVER (see Section 6.2) on the same OBDDs used to construct the MIPs. Note that instances for which SCOP SOLVER cannot compile the OBDD within the original cutoff time of 600 are not used in the experiments. The training set consists of 188 instances and the test set of 193 instances.

We used CPLEX `12.4` as MIP solver and `Gurobi 8.1.1` to create MIP instances from our OBDDs. We used the same protocol as described in Section 6.1.2 for this experiment, but use a cutoff of 300 seconds. This is because, compared to SCOP SOLVER, the modelling and compilation stage is skipped.

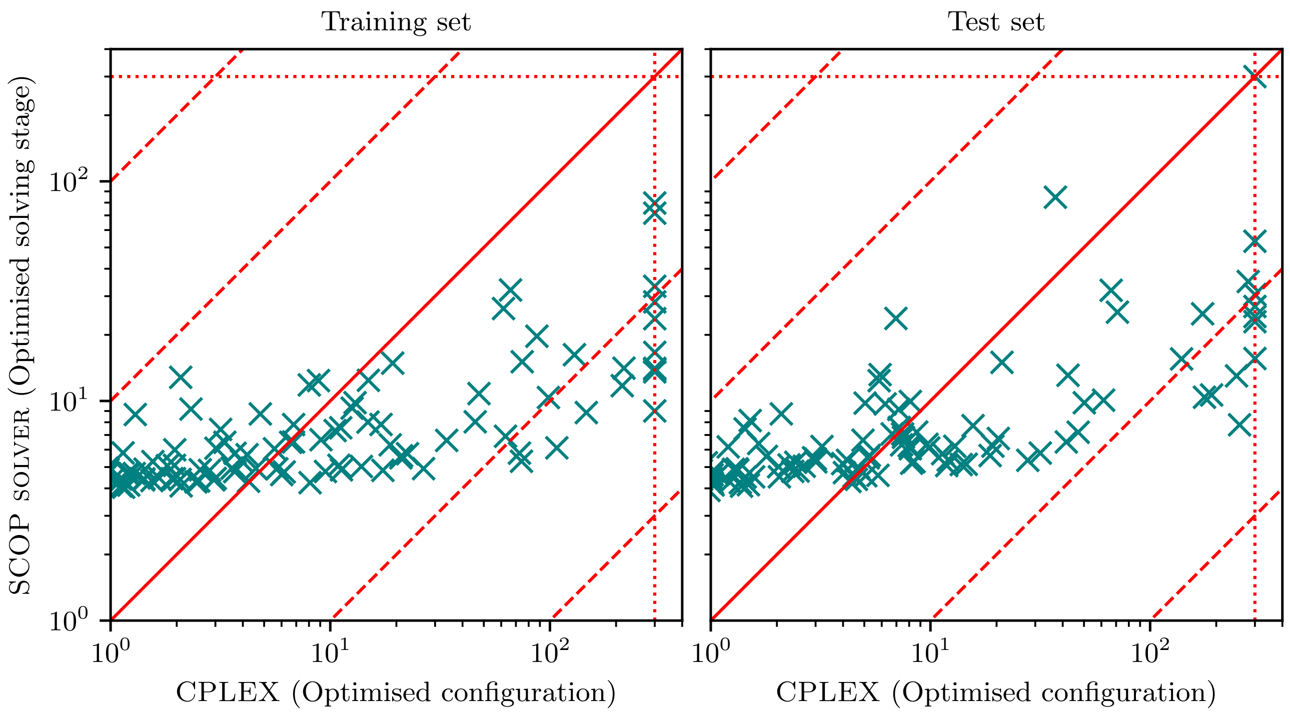
In Table A.4 we can see the resulting PAR10 values of the optimised and default configuration of CPLEX and the optimised configuration of the solving stage of SCOP SOLVER. The optimised configuration achieves a decrease of 28% compared to the default configuration on the training set and a decrease of 27% on the test set. In Figure A.2a we can see the running times on the individual instances and the number of timeouts of both configuration of CPLEX. In Figure A.2b the running times on the individual instances of the optimised configurations of CPLEX and SCOP SOLVER are compared. Even though the optimised configuration of CPLEX outperforms the default configuration, the optimised configuration of the solving stage of SCOP SOLVER still offers much better overall performance.

Table A.4: PAR10 values with a cutoff of 300 CPU seconds of both configurations of CPLEX and the optimised configuration for the solving stage of SCOP SOLVER on the instances from the viral marketing dataset.

Solver	Configuration	Training	Test
CPLEX	Default	170.4	166.4
	Optimised	122.0	121.3
SCOP SOLVER	Optimised	7.0	22.5



(a) Comparison of both configurations of CPLEX on both the training and test set. Both configurations have 7 timeouts on the training set. The optimised configuration has 7 timeouts on the test set and the default configuration 10.



(b) The optimised configuration of CPLEX compared to the optimised configuration of SCOP solver on both the training and test set. The latter does not have any timeouts on the training set and 1 timeout on the test set.

Figure A.2: Running time [CPU s] on individual instances, with a cutoff time of 600 seconds (indicated by horizontal and vertical dotted lines). The diagonal dashed lines represent differences in running time of factors of 10 and 100.

Bibliography

[1] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. In *Proceedings of IJCAI*, pages 733–739, 2015.

[2] K. Apt. *Principles of constraint programming.* 2003.

[3] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In *Proceedings of HM*, pages 108–122, 2007.

[4] M. Ben-Ari. *Mathematical logic for computer science.* 2012.

[5] V.D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *JSTAT*, 2008(10):P10008, 2008.

[6] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Proceedings of IWLS*, 1995.

[7] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers*, 45:993–1002, 1996.

[8] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *Proceedings of ACM-SIAM SODA*, pages 946–957, 2014.

[9] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[10] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[11] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of IJCAI*, pages 819–826, 2011.

[12] A. Darwiche and P. Marquis. A perspective on knowledge compilation. In *Proceedings of IJCAI*, pages 175–182, 2001.

[13] L. De Raedt, K. Kersting, A. Kimmig, K. Revoredo, and H. Toivonen. Compressing probabilistic prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.

[14] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: a probabilistic prolog and its application in link discovery. In *Proceedings of IJCAI*, pages 2468–2473, 2007.

[15] R. Drechsler, B. Becker, and N. Gockel. Genetic algorithm for variable ordering of OBDDs. *IEEE Computers and Digital Techniques*, 143(6):364–368, 1996.

[16] L. Duenas-Osorio, K.S. Meel, R. Paredes, and M.Y. Vardi. Counting-based reliability estimation for power-transmission grids. In *Proceedings of AAAI*, pages 4488–4494, 2017.

[17] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[18] I. Filippidis. dd python library, 2018. Available at `https://pypi.org/project/dd/`.

[19] L.C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[20] H.H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. 2011.

[21] H.H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[22] F. Hutter, H.H Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proceedings of CPAIOR*, pages 186–202, 2010.

[23] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of LION*, pages 507–523, 2011.

[24] F. Hutter, H.H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of ICML*, pages 754–762, 2014.

[25] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H.H. Hoos, and K. Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artificial Intelligence*, 243:1–25, 2017.

[26] F. Hutter, M. López-Ibáñez, C. Fawcett, M. Lindauer, H.H. Hoos, K. Leyton-Brown, and T. Stützle. AClib: A benchmark library for algorithm configuration. In *Proceedings of LION*, pages 36–40, 2014.

[27] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of IEEE ICCAD*, pages 472–475, 1991.

[28] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of ACM SIGKDD*, pages 137–146, 2003.

[29] A.L.D. Latour, B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, and S. Nijssen. Combining stochastic constraint optimization and probabilistic programming. In *Proceedings of CP*, pages 495–511, 2017.

[30] A.L.D. Latour, B. Babaki, and S. Nijssen. Stochastic constraint optimization using propagation on ordered binary decision diagrams. 2018.

[31] A.L.D. Latour, B. Babaki, and S. Nijssen. Constraint propagation on Binary Decision Diagrams for mining probabilistic networks. In *Proceedings of IJCAI*, pages 1137–1145, 2019.

[32] C.L. Lindner, G. Staudt, M. Hamann, H. Meyerhenke, and D. Wagner. Structure-preserving sparsification of social networks. In *Proceedings of ASONAM*, pages 448–454, 2015.

[33] M. López-Ibánez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, IRIDIA Technical Report Series, 2011.

[34] A. Nocaj, M. Ortmann, and U. Brandes. Untangling hairballs. In *Proceedings of GD*, pages 101–112, 2014.

[35] OscaR Team. OscaR: Scala in OR, 2012. Available at `https://bitbucket.org/oscarlib/oscar`.

[36] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of IEEE/ACM ICCAD*, pages 74–77, 1995.

[37] S. Panda, F. Somenzi, and B.F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of IEEE/ACM ICCAD*, pages 628–631, 1994.

[38] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming.* 2006.

[39] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[40] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of IEEE/ACM ICCAD*, pages 42–47, 1993.

[41] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *Proceedings of SIGMOD*, pages 721–732, 2011.

[42] F. Somenzi. CUDD: CU Decision Diagram package-release 2.4.0, 2004. University of Colorado at Boulder.

[43] B. Viswanath, A. Mislove, M. Cha, and K.P. Gummadi. On the evolution of user interaction in Facebook. In *Proceedings of ACM SIGCOMM WOSN*, pages 37–42, 2009.

[44] B. Wiegmans. Gridkit: European and North-American extracts, 2016.

[45] Y. Xue, X. Wu, D. Morin, B. Dilkina, A. Fuller, J.A. Royle, and C.P. Gomes. Dynamic optimization of landscape connectivity embedding spatial-capture-recapture information. In *Proceedings of AAAI*, pages 4552–4558, 2017.