7

Applying PbO to exact SCPMD solving

In the previous two chapters we presented several *stochastic constraint (optimisation) problem (SCP)* solving pipelines, based on stochastic constraint decomposition (Chapter 5) and global stochastic constraint propagation (Chapter 6). However, we did not explore in much detail how parameter settings affect the performance of our proposed methods, nor did we explore many alternatives for the design choices we made in the process.

We address these two open ends in this chapter, by applying the paradigm of *programming by optimisation (PbO)* to the methods described in the previous two chapters. Specifically, we implement and expose myriad alternative design choices for different elements of the solving pipelines, and then use *automated algorithm configuration (AAC)* to find application-specific optimised configurations of these pipelines. After configuration, we find that the global *stochastic constraint on monotonic distributions (SCMD)* solving pipeline from Chapter 6 outperforms its closest competitor (a *mixed integer programming (MIP)*-based decomposition pipeline from Chapter 5) on all test sets we considered by up to two orders of magnitude in terms of PAR10 scores. This chapter is based on the following peer-reviewed workshop paper and journal paper:

D. Fokkinga, A.L.D. Latour, M. Anastacio, S. Nijssen, and H. Hoos. 'Programming a Stochastic Constraint Optimisation Algorithm, by Optimisation'. In: *Data Science meets Optimization workshop 2019 (DSO 2019), colocated with IJCAI 2019, Macao, 2019.*

A.L.D. Latour, B. Babaki, D. Fokkinga, M. Anastacio, H.H. Hoos, and S. Nijssen. 'Exact Stochastic Constraint Optimisation with Applications in Network Analysis'. In: *Artificial Intelligence, vol 304, 2022.*

7.1 Introduction

As the results in Sections 5.3 and 6.5 show, different variants of solving methods behave differently on different problem instances. Based on this, we cannot decide what the optimal configuration is for each pipeline, or accurately predict how these or alternative configurations will behave on new problem types. Additionally, we largely relied on default parameter settings, with some minimal exploration of alternatives. Since generic solvers like Gurobi have many parameters, their defaults are unlikely to be optimal for a specific type of problem. While this might give us an indication of how well the decomposition method works 'out of the box', to assess its true potential, we need to tune its parameters. Finally, by learning which parameter settings yield shorter solving times for specific problems, we may also learn more about those problems and how to solve them more efficiently, potentially sparking interesting ideas for future research.

To address these observations, we leverage the PbO paradigm [80], which we briefly explained in Section 3.5. Specifically, in this chapter we make the following contributions:

1. We develop several design alternatives for different parts of decomposition-based and global constraint optimisation pipelines from Chapters 5 and 6 and expose them as configurable parameters (Section 7.2).
2. We apply AAC [79] to these configurable algorithms and demonstrate their effectiveness on benchmarks from two application domains (Section 7.3.3).
3. We then demonstrate how the optimised configurations of these methods generalise to harder problems and different problem settings (Section 7.3.4).

Automated optimisation techniques and tools such as SMAC [86] have been used to solve optimisation problems in approximate probabilistic inference [149], *constraint programming (CP)* solving [99] and MIP solving [85]. However, to the best of our knowledge, they have not yet been applied to the optimisation of the configuration of exact probabilistic inference methods.

In the remainder of this chapter, we first describe how we have applied the PbO paradigm to different elements of the *stochastic constraint (optimisation) problem on monotonic distributions (SCPMD)* solving pipelines described in Section 6.5.2. We then present our experiments in Section 7.3, and conclude this chapter in Section 7.4.

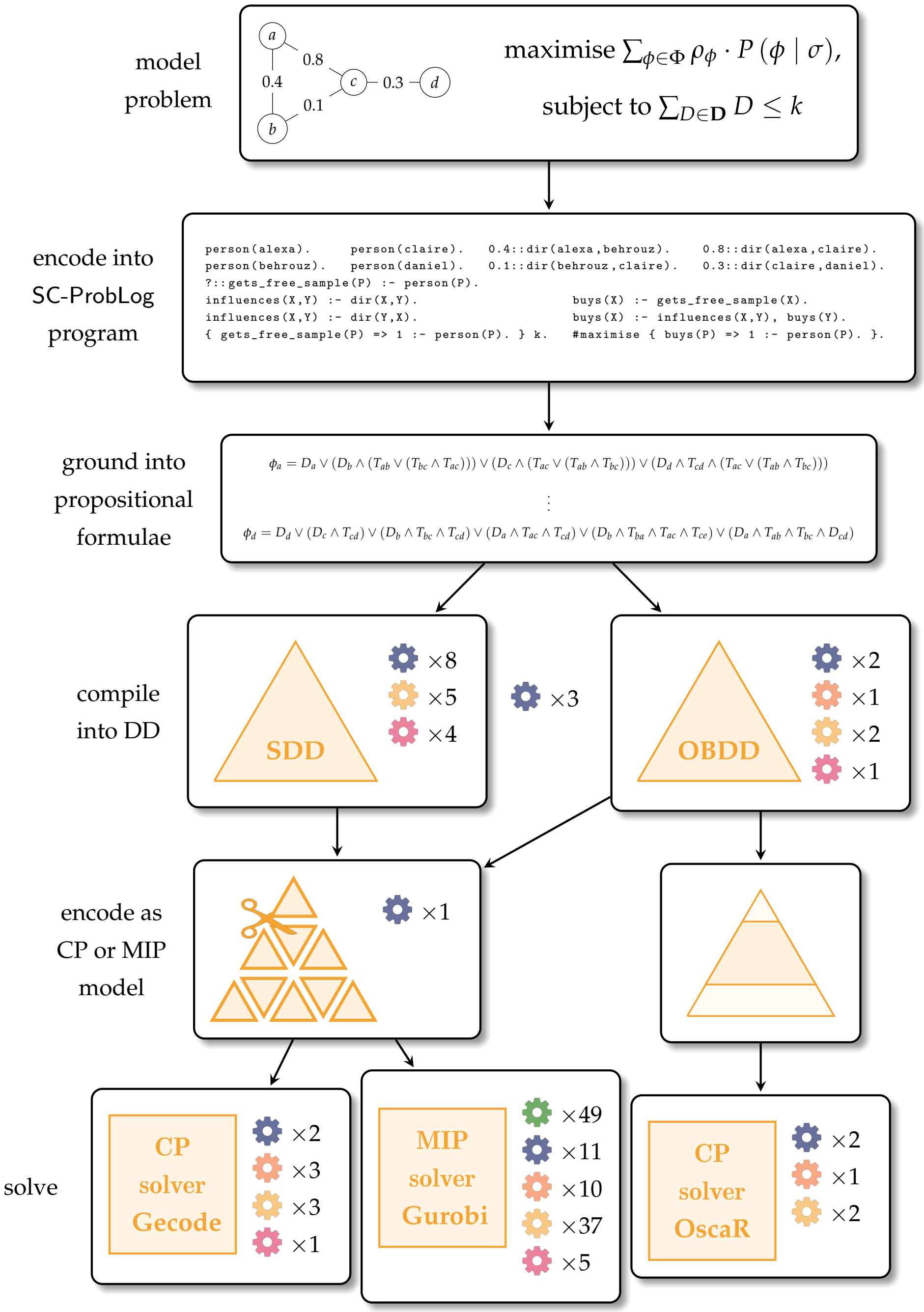
7.2 Design space of SCPMD solving pipelines

In Chapters 5 and 6 we described several approaches to solving SCPMDs. Figure 7.1 shows these different methods schematically, and visualises how they relate to each other. As we discussed above, in this chapter we apply the PbO paradigm on these methods. In this section we describe the different design choices that arise in the last three steps of the methods.

In this section we describe how we implemented alternative design choices where necessary, and how the addition of these design choices influences the size of the parameter space. As an illustration of the size of the parameter space in each step of the methods, Figure 7.1 shows the number of tuneable parameters, and their domain types, where applicable.

Note that we differ in our approach from earlier AAC approaches [85] by separating 'special values' of parameters from their regular domains. Parameters may have, for example, the domain of $\mathbb{N}^+$, but are turned off or tuned automatically when they take value 0. Contrary to earlier approaches, we split these parameters into a *switch parameter* and the *normal parameter*; the former turns the latter on or off, and the latter is only configured if the switch is on.

In the remainder of this section we discuss the different design choices available to us, or added by us, for the different solving methods in Figure 7.1. Note that Figure 7.1 shows the three pipelines as described in Section 6.5.2, with the one difference that the decomposition-based pipelines can now also compile the probability distributions to *sentential decision diagrams (SDDs)*, instead of only to *ordered binary decision diagrams (OBDDs)*. Details on the exact domains and the default values can be found at `github.com/latower/SCPMD-solving`.



```
person(alexa).     person(claire).   0.4::dir(alexa,behrouz).    0.8::dir(alexa,claire).
person(behrouz).   person(daniel).   0.1::dir(behrouz,claire).   0.3::dir(claire,daniel).
?::gets_free_sample(P) :- person(P).
influences(X,Y) :- dir(X,Y).                        buys(X) :- gets_free_sample(X).
influences(X,Y) :- dir(Y,X).                        buys(X) :- influences(X,Y), buys(Y).
{ gets_free_sample(P) => 1 :- person(P). } k.   #maximise { buys(P) => 1 :- person(P). }.
```

$$\phi_a = D_a \vee (D_b \wedge (T_{ab} \vee (T_{bc} \wedge T_{ac}))) \vee (D_c \wedge (T_{ac} \vee (T_{ab} \wedge T_{bc}))) \vee (D_d \wedge T_{cd} \wedge (T_{ac} \vee (T_{ab} \wedge T_{bc})))$$

$$\vdots$$

$$\phi_d = D_d \vee (D_c \wedge T_{cd}) \vee (D_b \wedge T_{bc} \wedge T_{cd}) \vee (D_a \wedge T_{ac} \wedge T_{cd}) \vee (D_b \wedge T_{ba} \wedge T_{ac} \wedge T_{ce}) \vee (D_a \wedge T_{ab} \wedge T_{bc} \wedge D_{cd})$$

Figure 7.1: *Overview of the different SCPMD solving methods that we evaluate in this chapter, along with an indication of the number of parameters that we configure in each step, with ⚙ switch, ⚙ Boolean, ⚙ categorical, ⚙ integer, and ⚙ real domains.*

7.2.1 Knowledge compilation step

The algorithms described in Chapter 5 take constraints on OBDD or SDD representations of probability distributions and decompose them into CP or MIP models. The algorithms proposed in Chapter 6 only operate on OBDDs representations. We summarise the parameters for the knowledge compilation step of our pipelines in Table 7.1.

Specifically, Table 7.1a shows the `Diagram` parameter, which we use to let the configurator choose between OBDD and SDD representations in the decomposition-based pipelines. For both types of *decision diagrams (DDs)*, we let the configurator tune if the knowledge compiler should attempt to minimise the diagram during compilation. Finally, the configurator can also tune whether the compilation algorithm should try to minimise the algorithm after compilation.

Table 7.1b shows the parameters related to minimisation in the CUDD 3.0.0 [168] compiler. In particular, we allow the configurator to exploit different minimisation algorithms: *Sifting (Sif)* [156], *Symmetric Sifting (SymSif)* [136], *Group Sifting (GSif)* [135], *Window Permutation (WP)* [89], a *Simulated Annealing (SA)* approach similar one from literature [21], a *Genetic Algorithm (GA)* inspired by one from the literature [58] and a randomised variable reordering algorithm based on the Sifting algorithm (*Rand*). They can be applied either to dynamic minimisation during the OBDD compilation, or we can make a call to a minimisation algorithm after compiling the OBDD.

The resulting parameter space for OBDD minimisation consists of two categorical parameters, one Boolean parameter, two integer-valued parameters and one real-valued parameter.

We use the SDD minimisation algorithm proposed in Section 5.2.2 as the only option for SDD compilation, as we must make sure that the resulting decomposed constraints can be linearised, for Gurobi to be able to find a solution in all cases. Otherwise, we expose all available parameters for configuration, resulting in eight Boolean parameters, five integer- and four real-valued parameters that can be tuned for the SDD minimisation.

One of these parameters, `ConvThreshold`, is used to define convergence: if the relative size reduction of the diagram is below this threshold, the algorithm has converged. All other parameters either limit the size of intermediate results, or the time that it takes to execute them, for the three vtree operations described in Section 5.2.2. Details are shown in Table 7.1c.

Combining these with the parameters in Table 7.1 results in a total of six categorical parameters, eight switch parameters, ten integer- and five real-valued parameters for the configuration of the compilation phase.

***Table 7.1:** The configuration space of the knowledge compilation step of our pipelines, in which the probability distributions are compiled into DDs. Note that some of the parameters are conditional on the value of other parameters. For brevity, we denote the Boolean domain of* {True, False} *with* $\mathbb{B}$*, and use k, M and B to indicate thousands, millions and billions, respectively.*

***(a)** Parameters for compilation in general.*

parameter	**domain**	**description**
`Diagram`	$\{OBDD, SDD\}$	Compile $\phi(\mathbf{D}, \mathbf{T})$ into an OBDD or SDD.
`DynMinimise`	$\mathbb{B}$	Use dynamic minimise during compilation (if `Minimise` $=$ *True*).
`Minimise`	$\mathbb{B}$	Minimise DD after compilation or not.

***(b)** Parameters for OBDD compilation with* CUDD*, all conditioned on* `Diagram` $=$ OBDD *and (*`Minimise` $=$ True *or* `DynMinimise` $=$ True*).*

parameter	**domain**	**description**
`VarOrder`	$\{Sif, SymSif, GSif, WP, SA, GA, Rand\}$	Variable reordering algorithm used for OBDD minimisation.
`Converging`	$\mathbb{B}$	Repeat variable reordering algorithm until no improvement on OBDD size is found (if `VarOrder` $\in \{Sif, SymSif, GSif, WP\}$).
`MaxSwap`	$[1, 3\text{M}]$	Upper bound on number of times two variables can be swapped in the variable order (if `VarOrder` $\in \{Sif, SymSif, GSif\}$).
`MaxSift`	$[1, 3\text{k}]$	Upper bound on number of variables that are sifted, *i.e.*, moved up or down in the variable order (if `VarOrder` $\in \{Sif, SymSif, GSif\}$).
`MaxGrowth`	$[0.0, 2.0]$	Upper bound on relative OBDD size increase during minimisation (if `VarOrder` $\in \{Sif, SymSif, GSif\}$).
`WSizes`	$\{2, 3, 4\}$	Evaluate permutations of `WSizes` consecutive variables in the variable order at a time (if `VarOrder` $= WP$).

***(c)** Parameters for for SDD compilation with the* `SDD` *package, all conditioned on* `Diagram` $=$ SDD *andand (*`Minimise` $=$ True *or* `DynMinimise` $=$ True*).*

parameter	**domain**	**description**
`ConvThreshold`	$[0.0, 50.0]$	Vtree convergence threshold.
`RRCartProdLimOn`	$\mathbb{B}$	Turn Cartesian product limit for right-rotate operations on.
`RRCartProdLim`	$[1, 65\,536]$	Maximum allowed size of a Cartesian product created by right-rotate operation (if `RRCartProdLimOn` = *True*).
`SWCartProdLimOn`	$\mathbb{B}$	Turn Cartesian product limit for swap operations on.
`SWCartProdLim`	$[1, 65\,536]$	Maximum allowed size of a Cartesian product created by right-rotate operation (if `SWCartProdLimOn` = *True*).
`RRTimeLimOn`	$\mathbb{B}$	Turn time limit for right-rotate operations on.
`RRTimeLim`	$[1, 25B]$	Time limit on right-rotate operation (if `RRTimeLimOn` = *True*).
`SWTimeLimOn`	$\mathbb{B}$	Turn time limit for swap operations on.
`SWTimeLim`	$[1, 25B]$	Time limit on swap operation (if `SWTimeLimOn` = *True*).
`LRTimeLimOn`	$\mathbb{B}$	Turn time limit for left-rotate operations on.
`LRTimeLim`	$[1, 25B]$	Time limit on left-rotate operation (if `LRTimeLimOn` = *True*).
`RRSizeLimOn`	$\mathbb{B}$	Turn size growth limit for right-rotate operations on.
`RRSizeLim`	$[1.0, 2.0]$	Size growth limit on right-rotate operation (if `RRSizeLimOn` = *True*).
`SWSizeLimOn`	$\mathbb{B}$	Turn size growth limit for swap operations on.
`SWSizeLim`	$[1.0, 2.0]$	Size growth limit on swap operation (if `SWSizeLimOn` = *True*).
`LRSizeLimOn`	$\mathbb{B}$	Turn size growth limit for left-rotate operations on.
`LRSizeLim`	$[1.0, 2.0]$	Size growth limit on left-rotate operation (if `LRSizeLimOn` = *True*).

7.2.2 Encoding step

We consider two main ways of encoding stochastic constraints. One is the decomposition approach presented in Chapter 5, the other is the global approach presented in Chapter 6.

For the decomposition approach, we take a constraint on a SDD or OBDD representation of the probability distribution, and either encode it into a MIP-model or a CP-model. For the CP-encoding of OBDDs specifically, we consider two variants: one that guarantees *generalised arc consistency (GAC)* (Section 6.2) and one that does not (Sections 5.2.1 and 5.2.2). Since one of the goals in this work is to develop an efficient SCMD propagation algorithm that guarantees GAC and uses an OBDD for the probability distribution encoding, we consider developing a GAC-guaranteeing CP encoding of stochastic constraints on probability distributions represented by SDDs to be outside the scope of this work.

The other main encoding approach is to keep the stochastic constraint as a global constraint on the OBDD representation of the probability distribution. For now, we consider only one such encoding. The accompanying propagation algorithm (Section 6.4) guarantees GAC by design.

Consequently, the encoding step actually only has one parameter, and only if we choose to model the probability distributions with OBDDs and then use the CP-based decomposition method to solve the problem: it determines whether we use the GAC-preserving encoding in this case, or not.

7.2.3 Solving step

In the following, we will briefly discuss the parameter spaces of the three solvers that we use in this work: Gecode, Gurobi and OscaR.

Solving with the decomposition method

For the methods that make use of Gecode and Gurobi to solve a linear program[1] obtained by decomposing a *stochastic constraint on probability distributions (SCPD)*, we enable the configuration of all parameters that are relevant for the speed of solving the problem exactly. We base the choices for domains and default values on earlier work on the automated configuration of Gurobi [85] and Gecode [99]. Considering the fact that Gecode and Gurobi already offer a wide range of branching heuristics, we refrained from exploring additional heuristics for these solvers.

[1]Even though Gurobi can handle quadratic constraints, we limit ourselves to linearised decompositions, as described in Section 7.2.1.

Table 7.2: *The configuration space of OscaR, when using the global SCMD propagator from Section 6.4. Some of the parameters are conditional on the value of other parameters.*

parameter	**domain**	**description**
`Sweep`	$\{Full, Partial\}$	Full or partial-sweep propagator.
`VarSelHeur`	$\{$*Top, Bottom, Derivative, Degree, Influence, Triangle, Similarity, Simmelian, ForestFire, Betweenness, Random*$\}$	Heuristics to select which variable to branch on next.
`ValSelHeur`	$\mathbb{B}$	Heuristics to select which value to branch on first.
`TimeSteps`	$[1, 1k]$	If `VarSelHeur` = *Influence*.
`NumSamples`	$[1, 100]$	If `VarSelHeur` = *Betweenness*.
`FireProb`	$[0.0, 1.0]$	If `VarSelHeur` = *ForestFire*.
`EdgesBurnt`	$[0.0, 1.0]$	If `VarSelHeur` = *ForestFire*.

The resulting configuration space for solving linear program encodings of SCPs with Gecode consists of two Boolean parameters, three categorical parameters, three integer- and one real-valued parameter. The configuration space of solving linear program encodings of SCPs with Gurobi consists of 49 switch parameters, 11 Boolean parameters, 10 categorical parameters, 37 integer- and 5 real-valued parameters. For practical reasons, we do not list the specific parameters here, but refer the reader to the above-mentioned repository for more details.

Solving with the global SCMD propagation method

Our experiments in Section 6.5 showed that branching order has an important impact on search efficiency. Because we study a variety of problems with different properties (Section 4.5), we decided to add a range of problem-specific branching heuristics to explore this result in more detail.

Table 7.2 shows the parameters for the global SCMD solving algorithm. Aside from the parameter to choose between using the full or partial-sweep algorithm, all parameters are directly related to branching.

The *Top*, *Bottom* and *Derivative* variable branching heuristics (with corresponding value branching heuristics) are described in Section 6.4.3. These heuristics are derived from the topology of the OBDD (in the case of *Top* and *Bottom*) or dynamically determined during the search (in the case of *Derivative*).

We propose seven new heuristics that take a different approach: they are derived directly from the probabilistic network on which the SCPMD is defined. An eighth new heuristic branches on variables that are selected uniformly at random.

In problems where decision variables are associated with nodes in a network (e.g., Example 4.2.1), the *Degree* heuristic branches based on the unweighted, undirected degree of the nodes. Similarly, *Influence* estimates the influence of nodes in order to quickly find a high-quality solution, inspired by work on social influence [23]. We translate the influence heuristic to problems with decision variables associated with the edges in the underlying network (e.g., Example 4.2.2), by taking for each edge the sum of the influence scores of its endpoints. We compute a degree-based score for edges using the *local-degree* measure from [109].

We observe that problems such as the theory compression problem of the **spine** problem instances or the power grid reliability problem of Example 4.2.2 are very similar to graph sparsification problems. We therefore derived the *Triangle*, *Similarity*, *Simmelian* and *ForestFire* heuristics from recent work on this problem [109, 163]. For the *Triangle* heuristic, we simply take the number of triangles that a node or edge is part of (not taking into account weights or directionality), to create versions of this heuristic that are suitable for problems with decision variables on nodes or edges, respectively. We translated the *Similarity*, *Simmelian* and *ForestFire* heuristics to problems with decision variables associated with the nodes in the underlying network (e.g., Example 4.2.1), by summing the scores of all incident edges on a node (not taking into account their weights or directionality). Finally, we use an estimate of either node or edge betweenness centrality as a proxy for the importance of a decision variable in the *Betweenness* heuristic.

Note that some branching heuristics incur preprocessing time, and that the computational complexity of this preprocessing as well as the quality of the resulting heuristic may depend on additional parameters. We mention these parameters in Table 7.2, but discussing them in detail is beyond the scope of this work. The resulting parameter space of the global SCMD propagator, consists of two Boolean parameters, one categorical parameter, two integer-valued parameters and two real-valued parameters.

7.3 Experimental evaluation

In this section, we report on experiments using AAC to determine which pipeline outperforms the others on two sets of problem instances, and to gauge how much each pipeline benefits from being automatically configured for these specific sets of problem instances. We first discuss the specific research questions that we are trying to answer. Next, we provide some details on the experimental setup in Section 7.3.2. Finally, we analyse the results of our experiments in Section 7.3.3, to answer the questions.

7.3.1 Research questions

The experiments in this section were designed to answer the following questions:

Q1 How much can we improve the performance of the decomposition methods and the global SCMD method on different real-world problems by automatically configuring these methods for those specific instance sets?

Q2 Which automatically configured method solves these problems best?

Q3 What can we learn about these solvers from the configuration results?

Q4 How do our optimised configurations generalise to harder instances of the same problem type and to instances of a different problem type?

7.3.2 Experimental setup

We briefly review the software, hardware and problem instances that we used for our experiments. In addition to the results in this section, the reader can find more, and more detailed, results at `github.com/latower/SCPMD-solving`.

Software and hardware

For our configuration experiments, we mostly used the software as described in Section 6.5.2. We used the NetworkX 2.2 and NetworKit 5.0.1 Python toolkits for computing the scores used for variable branching heuristics, as described in Section 7.2.3.[2]

SDDs were compiled using a version of the sdd 1.1.1 package [36] we adapted to generate SDDs that can be decomposed into linear programs, as described in Section 5.2.2.[3]

Because of the nature of the parameters described in Section 7.2, we expect that a model-based search process for optimal configurations will yield the best

[2]Available at `networkx.github.io` and `networkit.github.io`.

[3]Available at `reasoning.cs.ucla.edu/sdd/` and `github.com/ML-KULeuven/problog/tree/sc-problog`.

Table 7.3: *Summary of characteristics of the benchmark sets we used in our experiments. We provide the range of sizes of the set of interest $|\Phi|$, numbers of stochastic variables $|\mathbf{T}|$, numbers of decision variables $|\mathbf{D}|$, OBDD sizes $|OBDD|$ and the sizes of the training and test sets.*

name	**problem type**	$|\Phi|$	$|\mathbf{T}|$	$|\mathbf{D}|$	$|$**train**$|$	$|$**test**$|$
facebook	*spread of infl.*	15–30	16–107	15–30	412	411
high-voltage	*power grid rel.*	6–39	30–300	15–150	51	50

results. For our configuration experiments, we chose the general-purpose configurator SMACv3 [86], because it is one of the best-performing configurators that are model-based and freely available.

All experiments in this section were performed on GRACE, a cluster with 32 nodes, each equipped with 94 GB of RAM and two Intel Xeon E5-2683 CPUs with 16 cores, a cache size of 40 MB, running at 2.10 GHz using CentOS Linux 7.7.1908. Running times were measured in CPU seconds. We report on aggregated results by using *penalised average runtime with penalty factor 10 (PAR10)* values as a measure for running time performance.

In our experiments, we chose default values for compilation, CUDD, Gecode and Gurobi based on the literature [85, 99], on the results from Section 5.3, and on their own default settings. The default settings for OscaR were chosen based on the experiments in Section 6.5.3.

Benchmark sets

For automated algorithm configuration, we require a large set of instances. This is because we need disjoint training and testing of sufficient size for the configurator to learn from different instances (training) and then validate its performance on a sufficiently varied set of instances (testing). We created these instances using the processes described in Section 4.5 and summarise them in Table 7.3. All of the SCPMD instances we formulate on these problems are of **Variant 1** (see Section 4.5).

For the **facebook** benchmark set, we select all nodes in a problem instance as our set of interest. We choose the upper bound on the cardinality of the solution to be constant in these examples. Specifically, we use $k = 10$, because it can be expected to yield challenging problems, as seen in our results in Section 6.5. Additionally, fixing this threshold to one value, even for problems with different sizes, is a realistic choice for real-life applications in this setting. After all, com-

***Table 7.4:** PAR10 values in CPU seconds for the default (def.) and optimised (opt.) configurations of the three solving methods, for both the training set and the test set. We indicate in brackets the number of examples that hit our cutoff time (600 CPU s). We highlight the smallest PAR10 values on the test sets in* **bold**.

	CP-decomposition		MIP-decomposition		global SCMD	
	train	**test**	**train**	**test**	**train**	**test**
facebook (412 training instances, 411 test instances)						
def.	4 338 (295)	4 270 (289)	1 888 (124)	1 664 (108)	797 (52)	782 (51)
opt.	2 518 (168)	2 615 (174)	594 (39)	**627** (41)	751 (49)	682 (44)
high-voltage (51 training instances, 50 test instances)						
def.	4 386 (37)	4 351 (36)	3 686 (31)	3 989 (33)	2 379 (20)	2 782 (23)
opt.	4 379 (37)	4 452 (37)	3 188 (27)	3 031 (25)	2 260 (19)	**2 669** (22)

panies likely have a marketing budget that does not depend very directly on the size of the social network data they have access to.

We choose the threshold values for the **high-voltage** benchmark set differently. For these examples, we use $k = \lfloor |D|/2 \rceil$, such that we can reinforce at most half of the total number of power lines in any given problem instance. We believe this to be realistic for real-life applications, since we can assume that the maintenance budgets for power grids might be roughly proportional to their size.

7.3.3 Configuration results

To address **Q1** to **Q3**, we performed fifteen independent 48-hour runs of SMAC on each solving pipeline (Section 7.2), on the two training sets in Table 7.3, minimising the PAR10 (penalised average running time with penalty factor 10) and using a cutoff time of 600 CPU seconds. Then, for each method and each dataset, we evaluated the final incumbent (the configuration with the smallest PAR10 value) on the appropriate test set.

The results in Table 7.4 show that the MIP-decomposition method makes the largest relative improvement after configuration, which answers **Q1**. We explain this by noting that Gurobi has a relatively large configuration space (which gives many options for improvement), compared to Gecode and OscaR, and by noting that we used default settings for Gurobi as our default configuration, while we chose our default configuration of OscaR based on our results in Section 6.5.

We also observe that, even with configuration, the CP-decomposition method is not competitive with the MIP-decomposition method and global SCMD method, similar to what we see in Figure 6.4. Interestingly, for the CP-decomposition method, the automated configurator chooses the encoding that does not guarantee GAC for both the **facebook** and the **high-voltage** dataset. Once more, it appears that, for CP encodings of SCMDs, a global encoding is more favourable than a decomposed one. However, we see that the performance of the MIP-decomposition method and global SCMD method are comparable and complimentary after configuration, similar to what we see in Figure 6.5: on the **facebook** instances, MIP works better, while on the high-voltage datasets, the global constraint works better; this answers **Q2**.

We provide the optimised configurations obtained from these experiments online at `github.com/latower/SCPMD-solving`. To answer **Q3**, we first note that for the CP-decomposition and MIP-decomposition pipelines, SMAC always chooses to encode the probability distributions as OBDDs, rather than SDDs. Furthermore, here and in the global SCMD propagation pipeline, SMAC tends to favour the group sifting algorithm for OBDD minimisation, which is CUDD's default minimisation algorithm. Remarkably, the optimised configurations for the **facebook** and **high-voltage** sets agree on all parameter choices for OscaR: SMAC chooses to use the full-sweep version of the propagator, combined with the *Derivative-1* branching heuristic. We believe that further, detailed analysis of these and similar results of configuration experiments could provide useful directions for improvements to SCMD solving pipelines and see this as a promising direction for future work.

Finally, we note that the improvement in running time on the **high-voltage** benchmark set is less impressive (and even negative, in the case of CP-decomposition) than on the **facebook** benchmark set. We explain this by noting that the **high-voltage** example set is much smaller than the **facebook** set (and thus has fewer examples to learn from), while the problems tend to be larger (see Table 7.3), causing relatively many examples to hit the cutoff time.

7.3.4 Generalisation of automated configuration results

We addressed **Q4** by running the default and optimised configurations obtained from Section 7.3.3 on the examples in Table 7.3 that were not solved by any solver during the configuration experiment in Section 7.3.3 and therefore represent the hardest instances in the training and test sets. In this new experiment, however, we used a cutoff time of 3 600 CPU seconds instead of 600. Rather than using just one threshold k per problem instance, we ran each configuration with nine different thresholds per example, like we did for the experiments in Section 6.5.

In the configuration experiments, 19 of the **high-voltage** examples in the training set were never solved. Since we now use these examples again to evaluate the generalisation results, there is some leakage of information. However, since this is only $\frac{19}{351}$ and thus roughly 5% of the instances, we do not expect this to affect the results much. For the **facebook** set there are 5 such instances, which is less than 1%.

Similarly, we ran the optimised configuration obtained on the **facebook** dataset on the **hepth** and **facebook** examples described in Table 6.1, since these are spread-of-influence problems. Finally, we ran the optimised configurations obtained on the **high-voltage** dataset on the **spine** and **high-voltage** examples, because of their similarity. Note that, for practical reasons, there is a small overlap (at most 5%) in the instances used for training in Section 7.3.3 and the **facebook** and **high-voltage** test sets we are using in this experiment. We present the results in Table 7.5 and observe patterns similar to those in Table 7.4.

From Table 7.5a, we see that the results for the harder examples with the larger cutoff time are very similar to the ones shown in Table 7.4 and conclude that our configuration results translate predictably to harder instances of the same problem type, answering part of **Q4**.

Table 7.5b shows very similar results for the **facebook** and **high-voltage** problem instances. This is unsurprising, since they are taken from the same datasets and represent the same problem types as the ones used for the configuration experiments. For the **spine** and **hepth** examples, we notice dramatic improvement in the performance of the MIP-decomposition pipeline, but not so much for the global SCMD pipeline, with negative results for **hepth**. Still, the global SCMD pipeline outperforms the MIP-decomposition pipeline on these examples, with both the default and the optimised configurations. We conclude that the results obtained in Section 7.3.3 translate reasonably to problem instances of different types, with a small advantage for the global SCMD approach, answering the remainder of **Q4**.

7.4 Conclusion

In order to make a fair comparison between the SCMD solving pipelines proposed in Chapters 5 and 6, we applied the paradigm of PbO to these pipelines. This resulted in three highly configurable pipelines, with alternative design choices for the knowledge compilation, encoding and solving components. We used AAC to automatically configure these pipelines for instances from two different real-world application domains.

Our findings indicate that after configuration, the pipeline that encodes proba-

Table 7.5: *PAR10 values in CPU seconds for the default and optimised configurations on two test sets. We indicate in brackets the total number of (problem, k) combinations in the first column. In the other columns, we indicate in brackets how many of those combinations reached the cutoff time of 3 600 CPU seconds. For each problem set, we highlight the lowest PAR10 value for the optimised configurations in bold.*

(a) *Results for the problems in the benchmark sets in Table 7.3, that were not solved by any of the solvers in the configuration experiment of Section 7.3.3.*

		CP-decomp.	**MIP-decomp.**	**global SCMD**
facebook (558)	def.	35 398 (548)	28 780 (441)	11 330 (168)
	opt.	32 607 (504)	18 528 (278)	**10 716** (158)
high-voltage (351)	def.	34 325 (334)	33 523 (326)	29 300 (285)
	opt.	32 597 (317)	31 302 (304)	**29 186** (284)

(b) *Results on the full set of 52 examples in Table 6.1, with 9 threshold values for each example.*

		CP-decomp.	**MIP-decomp.**	**global SCMD**
spine (27)	def.	12 308 (9)	569 (0)	17 (0)
	opt.	16 220 (12)	35 (0)	**17** (0)
hepth (18)	def.	30 177 (15)	6 493 (3)	65 (0)
	opt.	26 254 (13)	568 (0)	**68** (0)
facebook (99)	def.	19 100 (52)	4 428 (11)	58 (0)
	opt.	15 482 (42)	791 (2)	**51** (0)
high-voltage (324)	def.	8 808 (77)	8 410 (74)	55 (0)
	opt.	8 538 (75)	4 447 (39)	**52** (0)

bility distributions as OBDDs and then solves the SCPMD using the global SCMD propagator proposed in Chapter 6 tended to outperform the other pipelines. This effect was particularly noticeable in the experiments in which we tested how well the optimised configurations generalise to larger instances, and to instances from different application domains. Note that this is also the pipeline that can only be applied to solving SCPMDs, and not to SCPs in general.

We also found that pipelines tended to favour OBDD representations of probability distributions over SDD representations and that a regret-based branching heuristic is always favoured for the SCMD propagation algorithm.

We applied AAC in the current study with a focus on running time minimisation. Other criteria can be of interest as well. For example, the memory use of knowledge compilers can be prohibitively large, so optimising solving methods to use less memory could increase the applicability of those methods. Furthermore, to the best of our knowledge, this work represents the first use of AAC in exact probabilistic inference. The configuration results we presented in this work encourage us to expect automated solver configuration to also be beneficial for optimisation solvers for other exact probabilistic inference tasks than the ones discussed in this work.