5

Decomposition methods for solving SCPs

Solving *stochastic constraint (optimisation) problems (SCPs)* requires efficient probabilistic inference and search. In this chapter, we propose a pipeline for effective SCP solving, which consists of two stages. In the first stage, we compile the underlying probability distribution of the input SCP into *decision diagrams (DDs)* (*ordered binary decision diagrams (OBDDs)* or *sentential decision diagrams (SDDs)*, specifically). These diagrams are converted into *arithmetic circuits (ACs)*, which are decomposed into models that are solved using *constraint programming (CP)* or *mixed integer programming (MIP)* solvers in the second stage. We show that, to yield linear constraints in those models, DDs need to be compiled in a specific form. We introduce a new method for compiling small SDDs in this form (OBDDs are naturally in this form). We evaluate the effectiveness of several variations of this pipeline on test cases in viral marketing and bioinformatics, and find that MIP-based methods outperform CP-based methods on all our test instances. This chapter is based on the following publication:

A.L.D. Latour, B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, and S. Nijssen. ‘Combining Stochastic Constraint Optimization and Probabilistic Programming — From Knowledge Compilation to Constraint Solving’. In: *CP*, Springer. pp. 495–511. 2017.

5.1 Introduction

Recall from Section 4.1 that this work is partially motivated by the observations that most SCP-related literature focuses on scheduling and planning problems, whereas SCPs formulated on probabilistic networks remain much less studied, and that there exists no generic language for programming stochastic constraint *optimisation* problems. We addressed both of these limitations in Chapter 4.

The aim of this chapter is to advance the state of the art in SCP solving on a third dimension, observing:

- There is no automatic pipeline for solving SCPs written in the new SC-ProbLog language.

We address this limitation by building a pipeline on technology that is taken both from the probabilistic reasoning literature and CP literature. For the probabilistic reasoning component, we leverage knowledge compilation technology, demonstrating how both OBDDs [26] and SDDs [46] can be used in this context, namely: by putting hard constraints on DD representations of probability distributions. In this chapter, our focus is primarily on SDDs, since they are known to lead to smaller representations of distributions than OBDDs [24] (see also Section 2.4).

We remark that, in this work, we study the use of compiling propositional formulae to OBDDs [26] and SDDs [46] to facilitate tractable *weighted model counting (WMC)*, in the context of SCP solving. Note that, in the *constraint programming (CP)* literature, both OBDDs and SDDs are often employed as compact representations for the satisfying assignments of a constraint [70, 77]. Here, we use these diagrams differently.

Specifically, we propose to convert the DDs into ACs, as described in Section 2.5, which we can use to compute conditional probabilities in a time that is linear in the size of the underlying DD. Part of the novelty of our approach lies in then formulating a hard constraint on the AC and *decomposing* that constraint (and thus the AC) into a set of local constraints.

We do this so we can translate a global constraint for which no propagation algorithm exists, into a set of constraints for which propagation algorithms have been developed and optimised for decades, to see how much we can benefit from these in the context of solving SCPs. Finally, we solve these constraints using CP and MIP technology.

Note that this *modular* approach to building an SCP solver has the advantage of allowing us to use the best building blocks for the pipeline that are on offer, instead of having to integrate different elements into one single solver. By using knowledge compilation as part of this pipeline, part of the model counting problem can be solved in a preprocessing phase, by the knowledge compiler. The resulting DD can then be passed on to the next phase, where it is used to enable repeated querying, which is useful in finding an optimal strategy, or finding a strategy that respects a certain constraint.

Another key technical contribution of this work is that we show that SDDs need to satisfy strict criteria in order for them to yield linear representations of probabilistic constraints. We introduce a new algorithm for minimising SDDs within this normal form. This allows us to reduce the size of the resulting ACs, while keeping the resulting constraint optimisation model *linear*, rather than *quadratic*, and thus easier to solve for MIP solvers.

The remainder of this chapter is organised as follows. In Section 5.2, we demonstrate how stochastic constraints on OBDD- and SDD-representations of probability distributions can be decomposed for solving with CP or MIP technology. In that section we also introduce the aforementioned normal form and our new SDD minimisation algorithm, as well as a solving pipeline based on DD decomposition. We present an experimental evaluation in Section 5.3, and conclude this chapter in Section 5.4.

5.2 Decomposing and solving stochastic constraints

In this section we describe the pipeline that we propose in this chapter in more detail.

Recall from Section 2.5 that, once we have compiled a probability distribution into a DD, we can transform that DD into an ACs to compute conditional probabilities. Specifically, we can use ACs to compute the success probabilities of residual probability-weighted propositional formulae $\phi|_\sigma$. A key observation is that the constraint in Equation 1.1 essentially is a constraint on the strategy σ. Recall that we encode probability distributions using DDs. Taking OBDDs as an example, we encode a strategy by adding weights to the outgoing arcs of the decision nodes. Thus, we can see Equation 1.1 as a constraint on the outcome of the AC that encodes $P\left(\phi|_\sigma\right)$, given the weights on the outgoing arcs of the probabilistic nodes of the underlying OBDD. We can also see Equation 1.1 as a constraint the weights we can put on the outgoing arcs of the decision nodes in that OBDD, to reflect a strategy. Because σ specifies value assignments to *Boolean* variables, we can cast solving Equation 1.1 as a *discrete* constraint satisfaction problem.

In this section, we demonstrate how we can decompose a constraint on an AC representation of a probability distribution into a (linear) program that can be solved by a CP or MIP solver. We first show how this can be done for ACs obtained from OBDDs, and then describe how we can do the same for ACs obtained from SDD representations. For the sake of brevity, we will often refer to "decomposing a constraint on an AC derived from a DD representation of a probability distribution" as "decomposing a DD".

We close this section with a proposal for a SCP solving pipeline that uses these DD decompositions.

5.2.1 Decomposing a stochastic constraint on an OBDD

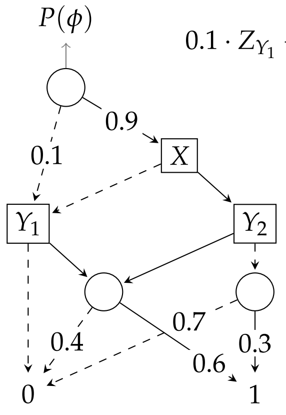
Recall from Section 2.5.1 that we represent a specific strategy by labelling the outgoing arcs of OBDD nodes labelled with decision variables with the values 0 and 1. Our aim is to solve Equation 1.1, which we interpret as a constraint on the values we can use to label those arcs. Therefore, we can interpret Equation 1.1 as a constraint on the AC induced by the OBDD that describes the probability distribution of an SCP.

Decomposition of a global constraint on an OBDD

We now show how we can decompose this global constraint on the OBDD into a multitude of smaller, local constraints.

Example 5.2.1 (Decomposition of a constraint on an OBDD representation of a probability distribution). *Figure 5.1 shows an example of an OBDD representation of a formula ϕ. We impose the constraint $P(\phi|_\sigma) \geq 0.4$. Figure 5.1 also shows an example of a decomposition of $P(\phi|_\sigma) \geq 0.4$ on the whole OBDD, adding auxiliary variables Z_{Y_1}, Z_{Y_2} and Z_X, whose domains include real numbers. This decomposition represents a CP or MIP model of the constraint $P(\phi_\sigma) \geq 0.4$.*

The next step to solving the global constraint, is to simply feed this set of smaller, local constraints to a CP or MIP solver. However, the decomposition in Figure 5.1 contains quadratic constraints, as illustrated in the following example, which are hard to solve for MIP solvers and CP solvers.



$$\begin{aligned}
P(\phi \mid \sigma) &\geq 0.4 \\
0.1 \cdot Z_{Y_1} + 0.9 \cdot Z_X &\geq 0.4 \\
Z_X &:= (1 - X) \cdot Z_{Y_1} + X \cdot Z_{Y_2} \\
Z_{Y_1} &:= 0.6 \cdot Y \\
Z_{Y_2} &:= 0.6 \cdot Y + 0.3 \cdot (1 - Y) \\
X, Y &\in \{0, 1\} \\
0 \leq P(\phi|_\sigma) &\leq 0.6 \\
0 \leq Z_X &\leq 0.6 \\
0 \leq Z_{Y_1} &\leq 0.6 \\
0.3 \leq Z_{Y_2} &\leq 0.6
\end{aligned}$$

Figure 5.1: *A small OBDD (left) with three stochastic variables (circular nodes) and two decision variables X and Y (rectangular nodes). The two nodes corresponding to decision variable Y are indexed for clarity. The decomposition on the right is constructed using Equation* 2.11 *(page 39).*

Example 5.2.2 (Quadratic constraint). *Figure 5.2 shows a graphical representation of the constraint in Example 5.2.1. It particularly shows a relaxation (recall the discussion of MIPs in Section 3.4) of the constraint. The coloured lines, labelled with values* 0.0 *to* 0.7 *represent contours on which the combination of (relaxed) X and Y values yield those particular values for* $0.1 \cdot Z_{Y_1} + 0.9 \cdot Z_X$, *and thus for* $P(\phi \mid \sigma)$.

We have added the extra constraint of $\sum_{D \in \{X,Y\}} D \leq 1$ *to the figure, resulting in a SCP that corresponds to the* constraint satisfaction problem (CSP) *in Example 3.3.2, and shaded the feasible region. It is easy to read from the figure that the only solution is* $(X = 0, Y = 1)$, *with value* 0.6, *as in Example 3.3.2*

Linearising quadratic constraints

Note that, while we can easily read the only solution to the SCP described above directly from Figure 5.2, the curved lines, due to the quadratic constraint, make it harder for the MIP solvers in particular to apply techniques such as branch-and-bound and cutting planes to narrow down the search towards integer solutions.

We therefore linearise this decomposition, for easier solving. A constraint of the form $A = B \cdot C$ can be linearised in the following cases:

1. At least one of the two variables in $\{B, C\}$ is a constant.
2. At least one of the two variables in $\{B, C\}$ is a Boolean variable.

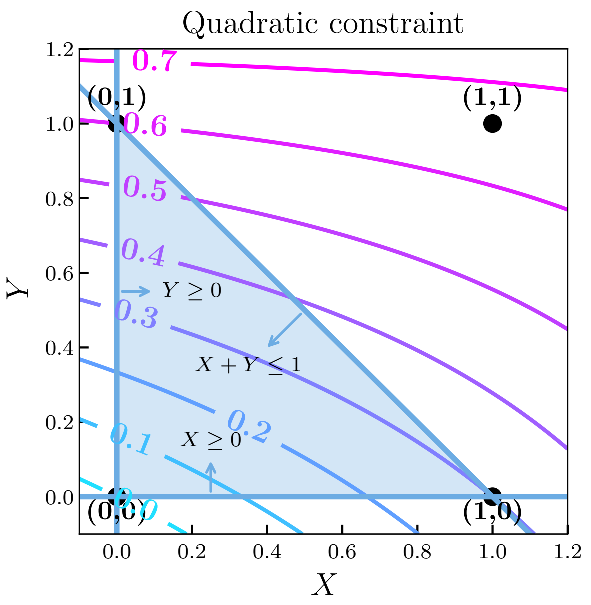


Figure 5.2: *Visualisation of the quadratic constraint* $0.1 \cdot Z_{Y_1} + 0.9 \cdot Z_X \geq 0.4$ *from Figure 5.1, as function of the values of* X *and* Y*. The added constraint* $X + Y \leq 1$ *makes this SCP correspond to the one described in Example 3.3.2.*

We obtain constraints that encode the OBDD by simply applying Equation 2.11, which we repeat here, for convenience:

$$v(r) := w(r) \cdot v\left(r^{+}\right) + (1 - w(r)) \cdot v\left(r^{-}\right),$$

to all (internal) nodes of the OBDD. An OBDD node r can be labelled either with a decision variable, or with a stochastic variable. If r is labelled with a decision variable, we can apply the big-M method [127] (with $M \leq 1$, because all real values are probabilities) to linearise the constraint expressed by Equation 2.11. If r is labelled with a stochastic variable, the arguments on either side of the '+'-sign each consist of a real number ($w(r)$ or $1 - w(r)$), multiplied by an expression ($v(r^{+})$ or $v(r^{-})$). If an expression is linear, multiplication with a real number preserves linearity. Summing two linear expressions yields another linear expression, making the constraint obtained by applying Equation 2.11 linear if r is labelled with a stochastic variable. Consequently, one of the above two cases always holds for all multiplications in the decomposition of a stochastic constraint on an OBDD representation of a probability distribution.

We illustrate this with the following example:

Example 5.2.3 (Linearising a quadratic stochastic constraint)**.** *The quadratic con-*

straint $Z_X := (1 - X) \cdot Z_{Y_1} + X \cdot Z_{Y_2}$ in Figure 5.1 (where X is a Boolean, as shown in Figure 5.1) can be linearised as follows:

$$\begin{array}{lll} Z_X := Z_{X_\top} + Z_{X_\bot} & Z_{X_\top} \leq X & Z_{X_\bot} \leq 1 - X \\ 0 \leq Z_{X_\top} \leq 1 & Z_{X_\top} \leq Z_{Y_2} + (1 - X) & Z_{X_\bot} \leq Z_{Y_1} + X \\ 0 \leq Z_{X_\bot} \leq 1 & Z_{X_\top} \geq Z_{Y_2} - (1 - X) & Z_{X_\bot} \geq Z_{Y_1} - X \end{array} \tag{5.1}$$

We linearise the model by repeating this method for all quadratic constraints.

5.2.2 Decomposing a stochastic constraint on an SDD

The decomposition of a constraint on a probability distribution represented by an SDD is very similar to that of a constraint on a probability distribution that is represented by an OBDD. One important difference is that not every SDD can be decomposed into a *linear* program. In the general case, SDDs yield *quadratic* programs, which are typically harder or impossible to solve for MIP solvers than linear programs. As we expect these constraints to be nonpositive semidefinite in the general case, we expect that we cannot apply *quadratically constrained quadratic program (QCQP)* solvers, either. We delegate the finding of a proof for this hunch to future work.

We first show why SDDs cannot be decomposed into linear models in the general case . Then, we identify a specific property of SDDs that does allow SDDs with that property to be decomposed into linear programs. Finally, we show how we can create an SDD minimisation algorithm for SDDs with this property.

From SDD to CP or MIP model

To see why we cannot linearise any decomposed constraint on an SDDs, recall the method for creating a linear model out of a constraint on an OBDD representation of a probability distribution as described in Section 5.2.1, and observe the difference between Equation 2.11 and Equation 2.12, which we repeat here for convenience:

Equation 2.11: $v(r) := w(r) \cdot v\left(r^+\right) + (1 - w(r)) \cdot v\left(r^-\right),$ (OBDD node), and

Equation 2.12: $v(r) := v\left(p^\ell\right) \cdot v\left(s^\ell\right) + v\left(p^r\right) \cdot v\left(s^r\right)$ (SDD node).

While constraints generated with Equation 2.11 can always be linearised (because $w(r)$ and $(1 - w(r))$ are either constants or Booleans), this is not the case for constraints generated with Equation 2.12. In that equation, the two arguments on either side of the '+'-sign are each a product of two expressions, e.g., $v\left(p^\ell\right)$ and

$v\left(s^{\ell}\right)$ on the left-hand side of the '$+$'. Even if those two expressions are themselves linear, their product can only be linearised efficiently if at least one of the expressions is constructed using only constants (i.e., weights corresponding to stochastic variables, in which case the product is trivially linear), or using only decision variables (in which case the product can be linearised using the big-M method).

We now identify a class of SDDs whose decompositions *can* be linearised.

Single-mixed path vtrees

Recall the description of vtrees in Section 2.4.3, and recall that they generalise the concept of variable order. Recall also that, while SDDs do not require a total order, we can derive a total order $\mathcal{O}$ from a vtree by traversing it in a left to right manner, noting the variables that label the leaves in the order in which they are encountered in this traversal. Different vtrees can thus correspond to the same total order $\mathcal{O}$.

We now show that it suffices to constrain the vtrees to ensure that the decomposition of the SDDs that respect them can be linearised. Recall that for each SDD decomposition node, the respected vtree determines the scopes of sub formulae represented by the prime and the sub. We observe the following: if all left-hand (right-hand) descendants of an internal vtree node n are stochastic variables, then for each SDD decomposition node (p, s) whose parent respects n, it holds that all variables occurring in n's prime p (sub s) are stochastic as well. A similar property holds for decision variables.

Recall from Section 5.2.2 that if the sub or the prime of a decomposition node represents a constant or a Boolean variable, this means that the constraints associate with those decomposition nodes can be linearised. Note that, if $sc(p) \subseteq \mathbf{T}$, the only variables in the scope of prime p are stochastic ones. Since stochastic variables can be considered as constants for the MIP model, we can precompute the corresponding value for the prime, effectively eliminating the MIP model variable associated with that prime. Similarly, if $sc(p) \subseteq \mathbf{D}$, the sub formula represented by a prime p consists only of decision variables, which can only take Boolean values in the decomposition. Since we can linearise all operations on Boolean variables [127], any prime containing only decision variables can be expressed by a Boolean variable with linear relations to other variables. Thus, in each of these two cases, the expression represented by the prime can be linearised and hence the product represented by the SDD decomposition node as well. The same holds for subs.

This leads us to define the concept of *mixed* and *pure* nodes in a vtree. A *pure* node is an internal node whose leaf descendants all are variables of the same type (either stochastic or decision), while a *mixed* node is an internal node that has leaf descendants of both types. We state that an SDD can be linearised into a MIP model if the vtree that it respects has the *single mixed path (SMP)* property.

Definition 5.2.1. *Given a vtree on variables of two distinct classes (e.g. decision and stochastic). This vtree has the* single mixed path (SMP) *property (and is called an* SMP vtree*) if, for each of its internal nodes n, the following holds: either both children of n are pure nodes, or one child of n is pure and the other child is mixed. As a consequence, if an SMP vtree has mixed nodes, all mixed nodes occur on the same path from the root of the vtree to the lowest mixed node.*

SMP-preserving SDD minimisation

Recall that SDDs that respect right-linear vtrees are equivalent to OBDDs. One can easily verify that a right-linear vtree has the SMP property: if it has an single mixed path, it is on the right spine of the vtree. From this follows that OBDDs can be linearised. However: right-linear vtrees generally do not yield the smallest SDDs. Since the size of the SDD determines the size of the resulting MIP model, and thus likely the solving time, small SDDs are preferable as input for the MIP model builder.

Choi and Darwiche have proposed a local search algorithm for SDD minimisation [36]. This algorithm considers three operations on the vtree: *right-rotate*, *left-rotate* (each well-known operations on binary trees) and *swap*. When a swap operation is applied to an internal node, the sub vtrees rooted at its children are swapped. Given a (sub) vtree, the greedy local search algorithm of Choi and Darwiche loops through its neighbourhood of different vtrees by applying consecutive rotate and swap operations, trying to find a vtree that yields a smaller SDD. Recall from Section 2.4.1 that we expect SDD minimisation to be $\mathcal{NP}$-hard.

Generally, this minimisation produces vtrees that do *not* have the SMP property, even if the initial vtree did, because rotation may remove this property.

A desirable property of Choi and Darwiche's algorithm is the following: the three local moves considered are sufficient to turn *any* vtree on a certain set of variables into *any* other vtree on the same set of variables. Consequently, the local moves in principle allow complete traversal of the search space of vtrees.

Here, we propose a simple modification of Choi and Darwiche's algorithm: we use the same local moves as their algorithm does, but any move that leads to a vtree that violates the SMP property is immediately rejected.

While this modification is conceptually easy, a relevant fundamental question

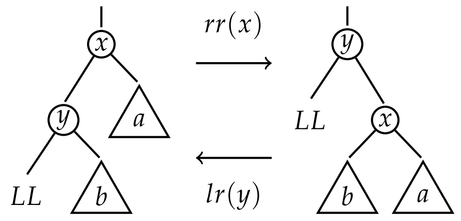


Figure 5.3: *Rotate operations on an SMP vtree. Node LL is the lowest variable in the variable ordering induced by these vtrees. Nodes x and y are internal; a and b are sub vtrees.*

is whether under this modification it is still possible to traverse the space of SMP vtrees on a fixed set of variables completely. We show that this is indeed the case.

In the following we refer to the leaf node that represents the variable that is lowest in the order associated with a vtree as LL (lowest leaf).

Lemma 5.2.1. *Let y be the parent and x the grandparent of the LL in an SMP vtree. Right rotate on x maintains the SMP property for the vtree rooted at y.*

Proof. Consider the left SMP vtree in Figure 5.3. Given that this vtree satisfies the SMP property by assumption, sub vtrees a and b cannot both be mixed, but one of them can be. Now consider the following cases:

Both a and b are pure and of the same class as LL: Lemma 5.2.1 holds trivially.

Both a and b are pure, not each of the same class as LL: Any class assignment to a and b will preserve the SMP property.

Node b is pure, node a is mixed: Since b is of the same class as LL (by assumption), node y is pure and node x is mixed. After applying right-rotate on node y, both y and x are mixed, and the SMP property is preserved.

Node b is mixed, node a is pure: Node a can belong to any class, since both node y and node x are mixed before as well as after applying right-rotate to y, preserving the SMP property under rotation.

These cases cover all possibilities for the classes of a and b. □

Note that the SMP vtree described above may be a sub vtree of a larger vtree. The fact that the right-rotate operation does not change the nature (mix or pure) of the root of this sub vtree, leads to the following corollary:

Corollary 5.2.1. *A right-rotate operation on the grandparent of the LL node does not change the SMP status of the full vtree.*

Lemma 5.2.2. *Given an SMP vtree, from which we derive total variable order $\mathcal{O}$ and a particular node LL. We can always obtain an SMP vtree from which we can derive the same total order $\mathcal{O}$, in which the LL is the left child of the root, through a series of right-rotate operations, without ever in the process transforming it into a vtree that violates the SMP property.*

Proof. A right-rotate operation on an internal vtree node decreases its left child's distance to the root of the vtree by one. Repeated applications of right-rotate on LL's grandparent ultimately makes LL's parent the vtree's root. By Lemma 5.2.1 and Corollary 5.2.1, the SMP status of the vtree never changes in this process. □

Lemma 5.2.3. *Given an SMP vtree on order $\mathcal{O}$, we can always obtain a right-linear vtree on the same order, through a series of right-rotate operations, without ever in the process transforming it into a vtree that violates the SMP property.*

Proof. By Lemma 5.2.2 we can turn any SMP vtree in one for which the LL is the left child of the root. This vtree can be made right-linear by recursively applying this method to the root's right child. □

Lemma 5.2.4. *A right-linear SMP vtree with variable order $\mathcal{O}$ can be transformed in any SMP vtree on the same variable order by a series of left-rotate operations without ever in the process transforming into a vtree without the SMP property.*

Proof. Since left-rotate is the dual operation of right-rotate, a sequence of right-rotate moves transforming any vtree to a right-linear one through right-rotate operations, can simply be reversed through left-rotate operations to turn a right-linear vtree in any other (on the same variable order). □

Note that rotate operations preserve the derived total order of the vtree, traversing the vtree from left to right, we still encounter the leaves in the same order. The only thing that changes, is the vtree's shape. However, the space of possible vtrees on a fixed set of variables is larger, since different total variable orders exist. The total order of variables is only changed by the application of swap operations.

Lemma 5.2.5. *Any right-linear vtree on variable order $\mathcal{O}$ can be transformed into a right-linear vtree on any other total variable order $\mathcal{O}'$ through a series of rotate and swap operations without ever in the process transforming into a vtree that violates the SMP property.*

Proof. Observe that any right-linear vtree satisfies the SMP property. Observe that if we can reverse the mutual total order of two adjacent variables (e.g. $A \prec B \prec$

$C \prec D$ becomes $A \prec C \prec B \prec D$), we can create any total variable order by repeatedly reversing the orders of adjacent variables. This reversal in the total order is simple to achieve. Suppose that node b in the right vtree of Figure 5.3 is a single variable, as is LL. We can make LL and b swap places by applying a left-rotate on y, resulting in the left vtree of Figure 5.3, and then applying a swap operation on y, followed by a right-rotate operation on x. □

Theorem 5.2.1. *Any SMP vtree can be transformed into any other SMP vtree on the same variable through a series of rotation and swap moves, without ever in the process transforming into a vtree that does not have the SMP property.*

We conclude that an SMP-preserving minimisation algorithm that applies only swap and rotate operations can in principle convert any SMP vtree into any other SMP vtree on the same variables. Note that, in principle, we could use an unrestricted minimisation algorithm. However, the search space of possible SMP vtrees on a given total order $\mathcal{O}$ is only a small part of the search space of all vtrees on $\mathcal{O}$. Therefore, it might not be easy or quick to transform a minimised SDD that violates the SMP property back into one that respects it, and we choose to adapt the minimisation algorithm in such a way that the vtree never loses the SMP property. Thanks to the above theorem, it is possible to traverse the entire search space of SMP vtree for a given total order, even though the path from one vtree to another might be very long.

Using the insights above, we implemented a greedy SMP-preserving minimisation algorithm as follows, building on the minimisation algorithm implemented in UCLA's sdd 1.1.1 library[1]. First, we compile an SDD without any minimisation. Since in the default settings, the resulting SDD respect a right-linear vtree, by Definition 5.2.1 this SDD has the SMP property. We then minimise this SDD by iteratively selecting an internal vtree node and exploring the neighbourhood of possible vtrees by performing SMP-preserving left-rotate, right-rotate and swap operations on it. We greedily choose that operation that reduces the size of the SDD the most. We repeat this process until the SDD size converges.

5.2.3 A decomposition-based SCP solving pipeline

In Sections 4.2 and 4.3 we showed how to model and program SCPs, and in Section 1.3 we identified two components to SCP solving complexity: probabilistic inference and search space traversal. Then, in Section 2.5, we showed how we can use the compact truth table representations that decision diagrams offer to

[1]Available at `reasoning.cs.ucla.edu/sdd`

tractably perform online probabilistic inference. We briefly argued for the use of CP and MIP technology for efficient search space traversal in Sections 3.3 and 3.4.

In this chapter we described how we can combine these ingredients to create constraint programs and mixed-integer programs that encode SCPs that can be solved efficiently.

In order to solve SCPs, we propose the following decomposition-based pipeline, see also Figure 5.4:

Step 1: Model the problem using a probabilistic network and (stochastic) constraint(s) or optimisation criterion.

Step 2: Program the problem using SC-ProbLog.

Step 3: Model the program for the queries present in the optimisation criterion of the SCP into a set of propositional formulae Φ.

Step 4: Compile a multi-rooted OBDD or SDD Δ, such that each root encodes the conditional success probability $P(\phi \mid \sigma)$ of one of the queries $\phi \in \Phi$, using possibly SMP-preserving minimisation algorithms for the SDD compilation to guarantee linearised models.

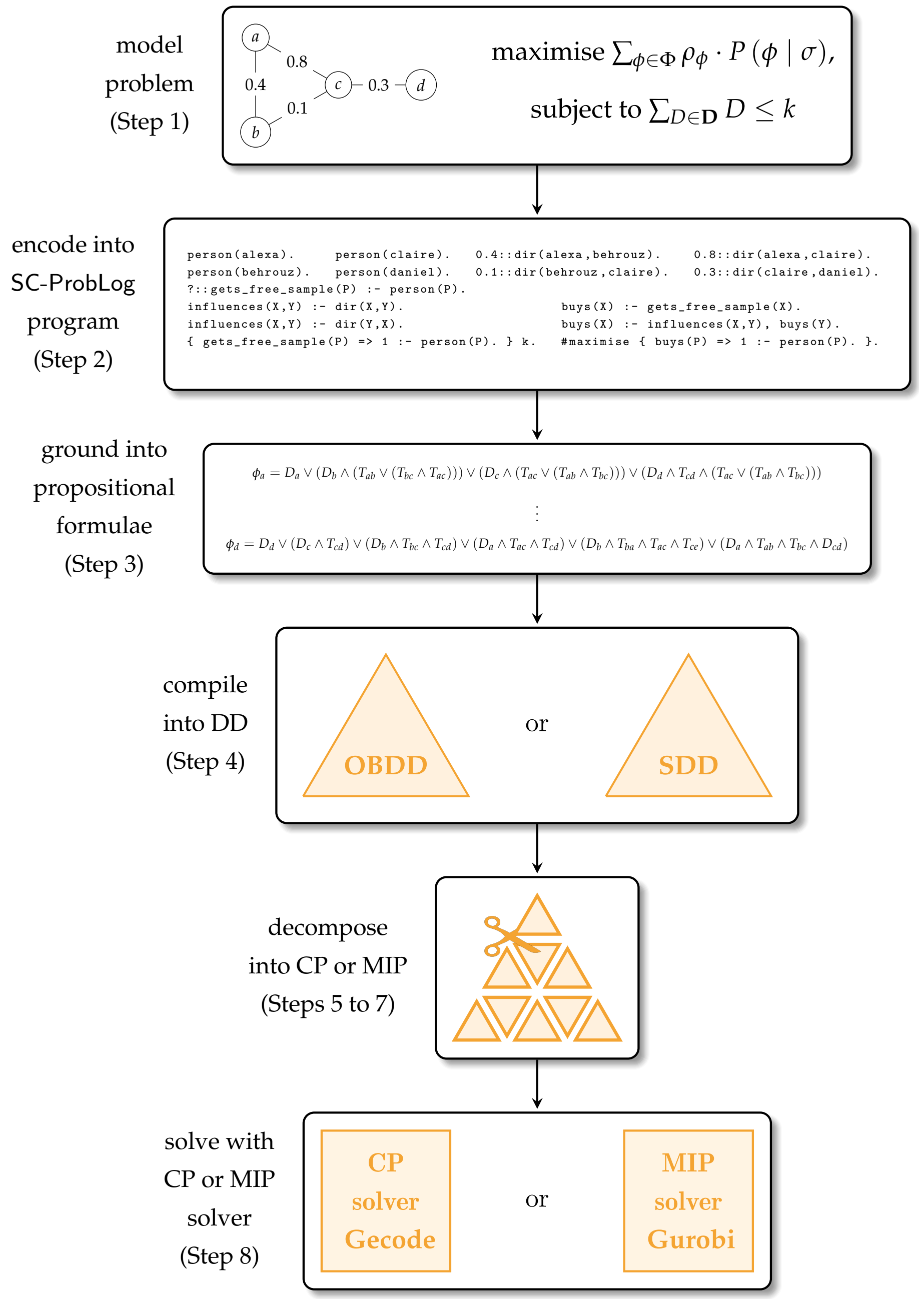
Step 5: Convert Δ into a multi-rooted AC (see Section 2.5), and then decompose this AC into a set of constraints, using the big-M method to linearise constraints when appropriate.

Step 6: Add the (stochastic) constraints to the set of constraints.

Step 7: Add the (stochastic) optimisation criterion to the resulting CP or MIP model.

Step 8: Use an off-the-shelf CP or MIP solver to find the optimal solution.

Note that, while we include OBDDs in the pipeline for reasons of generality, in this chapter, the focus in primarily on SDDs. Recall from Section 2.5 that SDDs that respect a right-linear vtree are actually OBDDs and that SDDs can be more succinct than OBDDs, once minimised. This motivates our choice to focus on SDDs in this chapter.



```
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 5.4: *Overview of the decomposition-based SCP solving pipeline we propose in this chapter.*

5.3 Experimental evaluation

We state some questions about the approach described in the previous section. Then we describe the experiments that we performed to answer these questions.

5.3.1 Research questions

Recall that the size of a MIP or CP model is linear in the size of the SDD representation of the probability distribution on which we impose a stochastic constraint. We expect smaller models to be faster to solve. However: minimising an SDD takes time. Furthermore, when quadratic constraints are allowed, we expect to obtain smaller SDDs; however, solving quadratic problems using CP may take longer than solving MIPs. We pose the following questions:

Q2 How do SDD sizes depend on the choice of minimisation algorithm?

Q3 How do the calculation times for the full toolchain compare for CP and MIP solvers, with and without appropriate minimisation?

Q4 How do the computation times for different phases of the algorithm compare to each other?

To answer these questions, and to demonstrate that SC-ProbLog programs can be solved in practice, we apply our algorithms to different SCPs. Of course, the constraints determine problem hardness, which begs the question:

Q1 Which threshold settings are useful for an evaluation of the solving times?

5.3.2 Experimental setup

We briefly describe our experimental setup and some details on the problem instances we used for our experiments.

Software and hardware

We implemented the stochastic constraint component of SC-ProbLog[2] in Python 3.4, building on the existing ProbLog 2.1 [59] implementation. ProbLog 2.1 uses UCLA's sdd 1.1.1 library [36], which is implemented in C, for SDD compilation.[3] We built on this code to implement our SMP-preserving SDD-minimisation algorithm. To convert constraints on SDDs representations of probability distributions into MIP models, we used Gurobi 6.52, which provides a convenient modelling

[2] Available at `github.com/ML-KULeuven/problog/tree/sc-problog`.

[3] Available at `reasoning.cs.ucla.edu/sdd`.

Table 5.1: *Some characteristics of the problem instances for the experiments in this section. We give the extracted community and variant of the problem we formulate on that network (see Section 4.5). We also provide the size of the set(s) of interest $|\Phi|$ and the number of decision variables $|\mathbf{D}|$ in the SC-ProbLog encoding of each problem. For each problem, we give the constraint threshold k or θ and objective value v_{obj} ('n/a' denotes a problem that has no solution for that threshold).*

instance	**problem type**	$\|\Phi\|$	$\|\mathbf{D}\|$	**threshold**	v_{obj}
spine16, variant 1	*sparsification*	23	33	$k = 15$	14.4
spine16, variant 2	*sparsification*	23	36	$\theta = 6.9$	8
spine27, variant 1	*sparsification*	13	76	$k = 25$	10.2
spine27, variant 2	*sparsification*	13	76	$\theta = 6.5$	8
spine27, variant 3	*sparsification*	26	86	$\theta = 1.3$	9.5
spine27, variant 4	*sparsification*	13	71	$\theta = 6.5$	52
hepth47, variant 1	*spread of influence*	20	20	$k = 10$	3.2
hepth47, variant 2	*spread of influence*	20	20	$\theta = 2$	6
hepth5, variant 1	*spread of influence*	10	33	$k = 20$	2.8
hepth5, variant 2	*spread of influence*	10	33	$\theta = 5$	n/a

interface through gurobipy.[4] We built our CP models using Gecode 5.0.0. We used Gurobi 6.52 as MIP solver and Gecode 5.0.0 as CP solver.[5]

We ran our experiments on a machine that we call JABBA. It has an Intel Xeon E5-2630 processor and 512GB RAM, running under Red Hat 4.8.3-9. For each individual computational step of the pipeline (Steps 3, 4 and 8) we used a timeout on our experiments of 3 600 s (1 hour).

Problem instances

For our experiments we use instances obtained from the **spine** [133] and **hepth** [131] datasets described in Section 4.5. We selected specific communities that we refer to as **spine16**, **spine27**, **hepth47** and **hepth5** in our results, and summarise some of the characteristics of the resulting problem instances in Table 5.1.

5.3.3 Results

To answer **Q1**, Figure 5.5 shows solving times for the **hepth47-v1** problem, for different thresholds. As expected, we find that thresholds that are not very strict

[4]Available at `www.gurobi.com`.

[5]Available `www.gecode.org`.

Table 5.2: *Performance in seconds of the different methods on the hardest instances (see Table 5.1) for the full pipeline. We show the solving times for SDDs obtained with compilation with no minimisation (t_{none}), with SMP minimisation (t_{smp}) and with default minimisation ($t_{default}$) for Gurobi and Gecode. We indicate a timeout with 't/o'.*

	Gurobi		Gecode	
instance	t_{none}	t_{smp}	t_{none}	$t_{default}$
spine16, variant 1	3.9	3.4	1 389.5	591.4
spine16, variant 2	4.1	3.9	70.9	31.4
spine27, variant 1	5.9	5.6	t/o	t/o
spine27, variant 2	4.7	5.7	t/o	1 878.2
spine27, variant 3	443.2	471.3	t/o	t/o
spine27, variant 4	23.3	21.9	222.9	8.6
hepth47, variant 1	545.8	412.7	t/o	130.9
hepth47, variant 2	188.6	163.8	2 859.9	6.9
hepth5, variant 1	2 076.8	1 185.7	t/o	t/o
hepth5, variant 2	364.6	346.4	t/o	t/o

or loose, require the longest solving times. We performed similar experiments for the other problem settings to systematically identify the threshold for which each problem was the hardest, which we then chose as test cases for the SCP solving method comparison.

To answer **Q2**, Figure 5.6 shows a comparison of the size reductions obtained by the SMP-minimisation algorithm and the default minimisation algorithm provided by the sdd library. We find that the SMP minimisation algorithm typically halves the size of the initial SDD. The default minimisation typically reduces the size of the SDD by one or two orders of magnitude.

To answer **Q3**, we summarise the performance of the four methods on our test cases in Table 5.2. For the **hepth5** problem we selected the ten highest-degree nodes for the queries, since the program could not be grounded within one hour if we selected all 33 nodes in the problem for querying. This reduced the grounding time to about 120 seconds. For the other test cases we have selected all queries in the problem, with grounding times in the range of 1–5 seconds.

We observe that without any minimisation of the SDD, Gurobi consistently outperforms Gecode. Furthermore, we observe that the difference made by SDD minimisation is larger for the Gecode methods than for the Gurobi methods. This can largely be explained by the results in Figure 5.6, and by those in Figure 5.7,

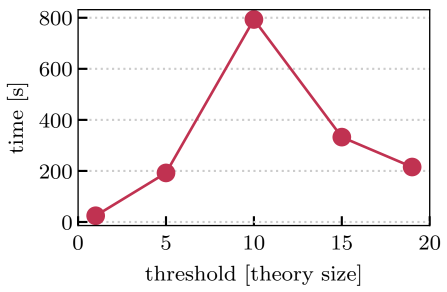


Figure 5.5: *Example of performance of Gurobi on a decomposed non-minimised SDD for different thresholds, for problem* **hepth47**, *variant 1.*

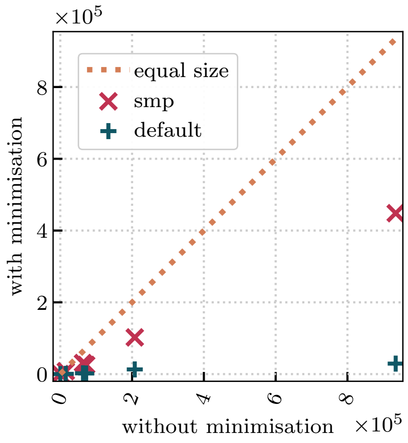


Figure 5.6: *Comparison of size reduction by SDD minimisation algorithms.*

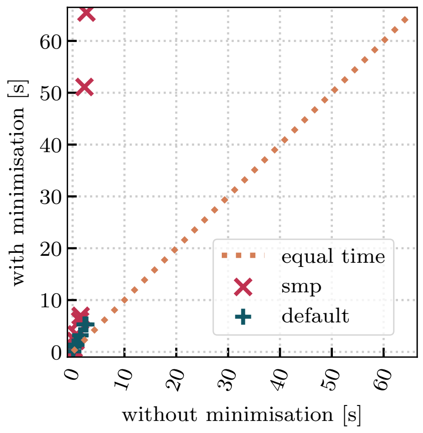


Figure 5.7: *Comparison of SDD compilation times.*

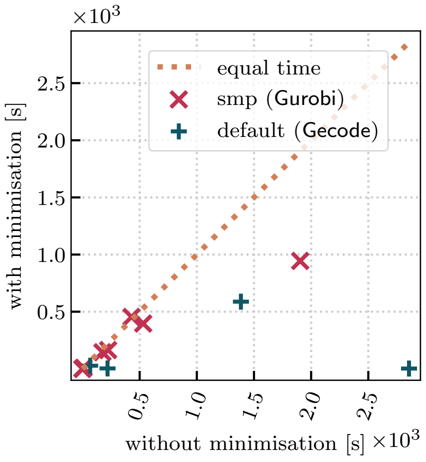


Figure 5.8: *Comparison of full pipeline solving times for the two solvers.*

which answers **Q4**. The latter results show that generally, compiling SDDs is a matter of seconds, whether they are being minimised or not. The exception is the **hepth5** problem, which takes tens of seconds to compile into an SDD when using SMP minimisation. Observe from the table that minimisation is still useful here, as it reduces solving time enough to make up for the extra minimisation time. We note that the minimisation algorithms are based on heuristics, and minimisation speed-up may lie in the improvement of these heuristics.

Finally, Figure 5.8 shows that the time that is gained during the optimisation part of the entire solving chain, can be orders of magnitude larger than the time lost by minimising the SDD. We do note that, since compiling the SDD can be done in seconds, this effect is less noticeable for the smaller problems.

5.4 Conclusion

In this chapter, and in Section 4.3, we showed how we can combine generic probabilistic programming technology (in the form of the SC-ProbLog programming language and knowledge compilation) and CP and MIP solvers (Gecode and Gurobi) to solve the type of SCPs that we described in Section 1.2. We combined these elements into a pipeline for solving these problems.

In constructing this pipeline, we presented two key contributions. The first is the decomposition of a hard constraint on an AC representation of a probability distribution (derived from its SDD representation), into a multitude of local constraints, such that they can be fed directly into an off-the-shelf CP or MIP solver.

The second key contribution in this chapter is the SDD minimisation algorithm that preserves properties that ensure that a constraint on an SDD representation of a probability distribution can be translated into a MIP model that is linearisable, while minimising the size of the SDD. This minimisation algorithm preserves a property of the vtree that defines the variable order of the SDD, which we call the SMP property.

In our experiments, we evaluated different variants of this pipeline on a range of problem instances from two different domains, exploring different combinations of stochastic constraints and optimisation criteria, and linear constraints and optimisation criteria. We showed that the pipeline that uses the MIP solver Gurobi consistently solved these instances faster than the pipeline that used the CP solver Gecode.

We also compared the running times of the pipelines when they use no SDD minimisation (which makes the compilation step fast, but results in larger models), or when they use SMP minimisation (in the case of the pipeline that uses Gurobi) or default minimisation (in the case of the pipeline that uses Gecode). Here we found that in both pipelines, minimisation consistently leads to shorter overall running times. In some cases minimisation makes the difference between a problem being solvable within the one hour time limit, or not. For the problems that are solvable within that time, minimisation can decrease the overall solving time with up to two orders of magnitude.

We note that a somewhat related study by Hemmi *et al.* also proposes a decomposition-based approach to solving SCPs [78]. There are, however, some important differences in both the scope and approach between their work and ours. While Hemmi *et al.* solve multi-stage SCPs, our focus is on single-stage ones. In multi-stage SCPs, the solution consists of a *policy* that dictates which decisions should be made in each stage as a scenario unfolds. Hemmi *et al.*'s methods solve multi-stage SCPs by generating all possible scenarios for the next stage, solving the SCP for each scenario, and continuing recursively. This decouples the stages from each other (which they call "relaxation"), and hence may cause constraints on decisions that span multiple stages to become decoupled. They address this by detecting which constraints are violated, pruning those partial solutions from the search space, and iteratively refine the solution.

Hemmi *et al.*'s approach is suitable for multistage problems, while ours only supports single-stage problems. However, since their method requires all scenarios to be generated, it can only handle small problems. While their experiments show results for problems with up to a total of 343 possible scenarios and 150 decisions (all solved within 800 seconds), the largest problem in our problem set has 6.7 million scenarios, and 86 decisions (solved within 444 seconds by our fastest method, but not solved within an hour by our slowest). Note that the approach of Hemmi *et al.* is highly parallelisable, and their results were run using a parallelised implementation, using 32 hyper threaded cores. In our experiments, Gurobi is the only solver that is parallelised and attempts to use as many threads as possible, which was 8 in our case.

While the results presented in this chapter are clearly encouraging, the methods we presented do have some weaknesses. Chief among them is that, in decomposing the constraint on the AC, we lose information about the structure of the underlying SDD. We expect that a dedicated propagation algorithm for such a constraint that exploits the structure instead of erasing it, may well outperform the CP-based implementation of the decomposition method, if such a propagator can be devised and implemented. The resulting method may then become competitive with the MIP-based decomposition method.