Universiteit Leiden
Opleiding Informatica

Incremental algorithms for solving stochastic constraint optimisation problems with probabilistic logic programming

Name:	Anna L. D. Latour
Date:	21/12/2016
1st supervisor:	Dr. Marcello M. Bonsangue
2nd supervisor:	Prof. Peter J. F. Lucas
external supervisor:	Dr. Siegfried Nijssen

MASTER'S THESIS

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

This work is the result of a collaboration between the Leiden Institute of Advanced Computer Science (LIACS) and the DTAI (*Declaratieve Talen en Artificiele Intelligentie*, or Declarative Languages and Artificial Intelligence) group at the Computer Science department of KU Leuven, Belgium.

It was supported by a grant from the Erasmus+ programme of the European Commission.

Abstract

Stochastic constraint optimisation problems (SCOPs) are constraint optimisation problems in which constraints are probabilistic, i.e., it is required that certain events happen with a bounded probability, as determined by a probabilistic model. In this work, we explore how probabilistic models implemented in probabilistic logic programming languages (PLPs) can efficiently be used when solving constraint optimisation problems. The naive way of solving SCOPs with PLP is by compiling optimisation and constraint models into large sentential decision diagrams (SDDs) during a preprocessing phase. This method is intractable for all but the smallest problems due to the compilation complexity of SDDs. We propose two methods that build SDDs for the models incrementally during search, thus eliminating the preprocessing phase and limiting the compilation times by keeping the SDDs small. We define a class of SCOPs on probabilistic social networks for which these incremental methods outperform naive methods, and show these method's effectiveness.

Contents

1 Introduction

Imagine you are Moriarty: a mathematics professor with too much time on their hands and a well-nurtured grudge against the world. You spend your days plotting elegant crimes that are executed by an intricate network of lesser criminals. Each of these crooks know only a few fellow villains, and are more likely to communicate with some of their contacts than with others. You have to protect your privacy, so you are not in direct contact with anyone. You have designed a sophisticated communication system that keeps your network of culprits in contact through encrypted links, which only you can control.

Familiar with the phrase 'keep your friends close, keep your enemies closer', you made sure that part of your network consists of these tiresome 'good guys', so you can keep an eye on them. However, you do not appreciate them getting too familiar with your plans...

Consider the following problem: you have designed an absolutely beautiful plan to destroy the world, but need to communicate instructions to one of your partners in crime. You'd prefer not to get in touch with them directly, so you consider sending the message through your network. Being the evil genius that your are, you designed the communication network for your thugs such that they can only communicate with their contacts if you open that link for them. Even if it is open, they might not forward information to their buddies, because they do not trust them at the moment, or might even believe that the information they received was false. Whatever the reason, you cannot predict what they will do, but you know with what probability they will pass your message along.

The question is: which links do you open in your criminal network such that the probability that your message reaches your accomplice is maximised, while the risk of one of the good guys hearing about it remains below a certain level?

That is an example of the type of problems we consider in this work: *Stochastic Constraint Optimisation Problems (SCOPs)* [30, 40], known also in literature as *Chance-Constrained Optimisation Problems (CCOPs)* [6].

Another example of a SCOP is the following. Suppose you have different advertisement campaigns for the same product, but targeted at different demographies. The best marketing is (word-of-mouth) viral marketing, where people recommend your product to other people, because of the reasons that are prevalent in you different add campaigns. Who must you target such that the probability of the campaign messages reaching the correct demographic is maximised, while keeping the probability that it reaches other demographics is limited (because this might have an adverse effect)?

Particularly, we consider problems that are represented by graphs in which (some of) the edges are probabilistic. We choose these types of problems because we believe that many real-world problems can be defined on probabilistic graphs, such as problems in bioinformatics, viral marketing, and epidemics and pandemics.

We study optimisation criteria and constraints that involve probabilities for paths in probabilistic graphs. The existence of a certain path in such a graph can be expressed in logic formula, that is true with a certain probability. We consider the optimisation problem of selecting a subset of the probabilistic edges, such that the probability for a certain path is optimised. Other paths in the graph are used as probabilistic constraints: the probability for some of these paths has to stay above or below a certain threshold. We also consider non-probabilistic optimisation criteria, e.g. maximising the number of selected probabilistic edges, and problems with additional probabilities, such as uncertainties for the existence of nodes in the network.

Solving these types of SCOPs, where the constraints are expressed as bounds on the probabilities of logic formulas, requires a general programming system in which those problems can be encoded. In this work we propose to combine *Probabilistic Logic Programming (PLP)* with *Constraint Programming (CP)*, allowing for the formulation of problems in which distributions are subject to constraints. In this work we present such a combination, and introduce methods for solving SCOPs.

We observe that we need only compute exact values for the optimisation criterion, while mere bounds for the probabilities of the constraint paths are generally sufficient to solve the problem. We use this notion in a proposal for an efficient method for solving Constraint Optimisation Problems (COPs) with probabilistic constraints in general, and the one described above in particular.

We believe that many real-world problems, such as gene activation problems, the limiting of viral outbreaks, and viral marketing problems, can be formulated as SCOPs. Even more problems can be formulated in a SCOP if the programming system used for solving them accommodates utilities as well as multiple optimisation criteria. The contribution of this work consists of methods that can be part of such a programming system.

Given the problem, combining CP with a PLP language seems natural. In the next chapter we discuss a number of these languages. We introduce the language of our choice, (DT)ProbLog in Chapter 3. We formalise the SCOP and its various settings in Chapter 4, where we will also introduce a number of other example problems. Our proposed methods for solving these kinds of problems are presented in Chapter 6. We continue with a description of our experimental setup in Chapter 7, after which we present and discuss the results of these experiments in Chapter 8. Our conclusion and suggestions for future work can be found in Chapters 9 and 10, respectively. At the end of this document a list of symbols and acronyms is provided. It is followed by the appendices, which contain a brief introduction to propositional logic and first-order logic (Appendix A), as well an introduction to knowledge compilation (Appendix B).

2 Related Work

The field of PLP has a quarter-century long history, starting with David Poole's introduction of a Prolog (see Chapter 3.1) extension for probabilistic Horn abduction in 1991 [23], which could express any model-based knowledge and any probabilistic knowledge that could be represented by a Bayesian belief network. In 1995, Taisuke Sato introduced a statistical learning method for probabilistic programs, integrating symbolic computation (logic programs) with a probabilistic framework (statistical modelling) on the semantic level by using distribution semantics [31]. Distribution semantics add a random variable to probabilistic facts and rules in a logic program, or Knowledge Base (KB). New (statistical) knowledge can be derived from these probabilistically annotated facts and rules. The use of distribution semantics is typical for most probabilistic logic programming languages.

This statistical learning method evolved into the symbolic-statistical modelling programming language PRISM [32], in 1997. The PRISM tool learns (the parameters of) statistical models from a logic program and a set of observations. The tool is still being updated, the latest version was released in March 2016 [33]. PRISM is built on B-Prolog, a logic constraint programming implementation of Prolog [34].

The field of learning stochastic models from examples and logic programs is called Statistical Relational Learning. For an overview, see *Statistical Relational Artificial Intelligence*, De Raedt et. al. [14].

Following these methods for learning logical models from probabilistic knowledge, in 1995 a logical programming language for (multi-agent) decision making under uncertainty was introduced by David Poole: Independent Choice Logic (ICL) [25]. This offspring of the probabilistic Horn abduction is an extension of the latter in the sense that where probabilistic Horn abduction only allows a logic program to compute the consequences of probabilistically independent 'choices', ICL allows for multiple agents to make their own choices. ICL also allows *negation as failure* in the logic: if ICL fails to find a proof that shows a statement to be true, this statement is interpreted as being false [24].

Other probabilistic logic programming languages include P-log [2], Bayesian Logic Programs [20], Logic Programs with Annotated Disjunctions (LPADs) [37].

In 2006 Matthew Richardson and Pedro Domingos proposed Markov Logic Networks (MLNs) as a combination of first-order logic and probabilistic graphical models in a single representation [26]. MLNs can perform inference to answer conditional queries on probabilistic logic programs ('what is the probability that formula f_1 is true, given that formula f_2 holds?'), known in literature as the MARG task (for *marginal* probability). Another task that is supported by MLNs, is that of finding the most likely state of a set of variables, known as MAP (*Maximum a Posteriori*). The inference is performed by Markov chain Monte Carlo (MCMC). Note that the MAP task is an optimisation task: it can be used to find which binary values variables should have in order to maximise the probability for a certain formula to hold. However MLNs do not support hard constraints, and are as such less suitable for the kind of SCOP considered in this work.

For an overview of PLP methods, see *Probabilistic (Logic) Programming concepts*, De Raedt et.al. [12].

The probabilistic logic programming problems regarded in this work, however, are implemented using ProbLog, a probabilistic extension of Prolog that was proposed by Luc De Raedt et.al. in 2007 [13]. It is a probabilistic extension of Prolog that uses Sato's distribution semantics, and negation as failure (as does Prolog). ProbLog 1 was originally proposed as a solving method for link discovery in graphs that represent biological data. The link discovery task involved computing the probability of there being a path from one node in the network to another, where the edges are represented by (independent) probabilistic random variables.

This explains our choice for ProbLog as the PLP language for this work: it was designed for the computation of the probability of the existence of paths in probabilistic graphs.

Note that PLP languages generally focus on computing the success probability of a query (e.g. 'what is the probability that there is a path from node a to node b?'). Programming paradigms such as graphical models (e.g. Markov (Logic) Networks and Bayesian Networks) focus more on tasks such as computing the marginal probability of a (set of) random variables, given some evidence (the MARG task mentioned above), or on finding the most likely joint state of a set of random variables (*Most Probable Explanation*, or MPE). When ProbLog 2 was presented in 2013, it also supported the last two inference tasks [17], making it a versatile logic programming tool. The MAP task is also supported by ProbLog.

In the next chapter we provide an introduction to the syntax, semantics and inference methods of ProbLog.

While the history of PLP dates back to the early nineties, the dawn of CP took place during the sixties, when Golomb and Baumert published a backtracking search method [18] (backtracking is the

most widely used search method in CP). During the following twenty years, CP research was split into two tracks that dominated that particular branch of artificial intelligence: the language track and the algorithmic track [27, ch. 2].

In 1972 the language track brought forth Prolog [9, 27], a language that can be seen as a constraint programming language. The algorithm track focused mostly on Constraint Satisfaction Problems (CSPs), for example with Alan Mackworth and Eugene Freuder publishing an analysis of the complexity of consistency algorithms (see Chapter 3.4) for the solving of CSPs, in 1985 [21].

Ever since, the field of constraint programming has been well-established in AI, with applications in job scheduling, packing problems and viral marketing.

In the next chapter we first provide introductions to Prolog, ProbLog and a deterministic PLP language: DTProbLog. We conclude the next chapter with an introduction to some basic solving techniques that are widely used in CP.

To the best of our knowledge, the combination of CP and PLP for the particular types of SCOP, that are described in more detail in Chapter 4, have not yet been studied in literature.

3 Background

In this chapter we provide an introduction to the logic programming language *Prolog*, its probabilistic extension *ProbLog* and finally its decision-theoretic extension *DTProbLog*.

We assume that the reader is familiar with both propositional logic and first-order logic. Please refer to Appendix A for a brief introduction to these logics.

3.1 Prolog: reasoning with facts and rules

The declarative programming language Prolog was developed by Alain Colmerauer and Philippe Roussel during the early 1970s [9]. Where *imperative* programming languages require the user to specify how the computer should solve a problem, *declarative* programming languages allow the user to describe all that is known about the problem, and query the computer to find answers to questions, without having to worry about how those answers are obtained exactly. This makes declarative programming languages generally easy to learn compared to imperative programming languages.

In the next chapters, we briefly describe the syntax and semantics of Prolog, after which we will discuss the way in which Prolog performs logical inference. A more detailed description of Prolog can be found in *Learn Prolog Now!*, by Patrick Blackburn, Johan Bos and Kristina Striegnitz [4].

Syntax and semantics: querying a knowledge base

As stated before, Prolog allows the programmer to specify all he/she knows about the problem under consideration. This knowledge is represented in a *knowledge base* (KB) as *facts* (mostly in the form of *relations* or *predicates*) and *rules*. An example of a fact is:

```
vulcan(spock).
```

which is a relation that says that `spock` is a `vulcan`. Note that `spock` and `vulcan` are *constants*, whose names always begin with a lowercase letter in Prolog. The predicate `isa` also starts with a lowercase letter, as do all predicates in Prolog. Note that lines in Prolog always need to end with a period (`.`). An example of a rule is:

```
actsRationally(X) :- vulcan(X); borg(X).
```

A rule consists of a *head*, which is the part left of the `:-` mark, and a *body*, which is the part right of the mark.[1] The `X` in the rule above is a variable, and thus is written with a capital letter. This rule says that if the constant we substitute for `X` is a Vulcan or a Borg, then the person represented by this constant acts rationally. Prolog uses the '`;`' symbol to express a logical disjunction, and the '`,`' symbol to express a conjunction. Logical disjunctions can also be expressed in a different way. The rule above may also be written as:

```
actsRationally(X) :- vulcan(X).
actsRationally(X) :- borg(X).
```

For conjunctions on the other hand, there is only one way of expressing them:

```
canDoMindMelds(X) :- vulcan(X), trained(X).
```

Once we have a logic program, we can use Prolog to *query* that program in order to answer questions about our problem, such as:

```
?- canDoMindMelds(spock).
```

which asks if Spock can do mind-melds, or:

```
?- actsRationally(X).
```

which asks Prolog to provide all the people specified in the knowledge base that act rationally. How Prolog answers these queries is discussed in the next chapter.

A commonly used programming construct in Prolog is that of *recursion*, which can be used to formulate problems that describe paths in graphs. An example of such a program is the following:

[1] Note that the fact above is actually shorthand for the following rule:

```
vulcan(spock) :- True.
```

Program 1: Brexit.

```
train(london,paris).
train(paris,brussels).
train(brussels,paris).
train(brussels,amsterdam).
train(amsterdam,brussels).
plane(london,amsterdam).
plane(london,brussels).

connection(X,Y) :- train(X,Y).
connection(X,Y) :- plane(X,Y).
path(X,Y) :- connection(X,Y).
path(X,Y) :- path(X,Z), connection(Z,Y).
```

This program represents ways to get from one city to another, by using train and air plane connections. One can get from one city to another if there is a path between those cities. A path from `X` to `Y` is either a direct air plane (`plane(X,Y).`) or train (`train(X,Y).`) connection, or a change of such connections. Note that the paths are directed, and that for this specific example, it is impossible to get to London, while it is possible to get out of there. We can use this program to ask queries such as 'list all possible destinations if I leave from London':

```
?- path(london,Y).
```

Note that Prolog operates under the *closed world assumption*, which says that facts that are specified in the knowledge are true, and anything that is not in the knowledge base, is false. Thus, in the example above, because there is no train specified from, for example, Paris to London, Prolog assumes that such a train does not exist. However, Prolog also allows for negation in order to create rules for situations in which something is specifically not true. Other Prolog features include operations on numbers and lists, but these are all out of the scope of this work. For more information on the use of these features, consult *Learn Prolog Now!* [4].

Inference: matching and proof search

So how does Prolog find the answers to the queries it is presented with? This is done through the process of *term matching*. Consider the following program:

Program 2: Pride & Prejudice.

```
f1      friendof(elisabeth,jane).
f2      friendof(caroline,jane).
f3      friendof(mrbingly,caroline).
f4      friendof(mrdarcy,elisabeth).
f5      friendof(mrdarcy,mrbingly).
l1      likes(X,Y) :- friendof(X,Y).
l2      likes(X,Y) :- friendof(X,Z), likes(Z,Y).
```

(with the lines labelled for reasons that will become clear) and suppose we pose the query:

```
?- likes(mrdarcy,jane).
```

where we ask Prolog if Mr. Darcy likes Jane. Prolog will search for a way to *match* (or *unify*) the term `likes(mrdarcy,jane)` with a term in the knowledge base. In this work, we will use the (loose) definition of matching from *Learn Prolog Now!*:

Definition 1. *Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.*

Thus, term `elisabeth` matches with `elisabeth` (they are the same constant), term `friendof(mrdarcy, elisabeth)` matches with `friendof(mrdarcy,elisabeth)` (they are the same predicates) and variable `X` matches with the same variable `X`.

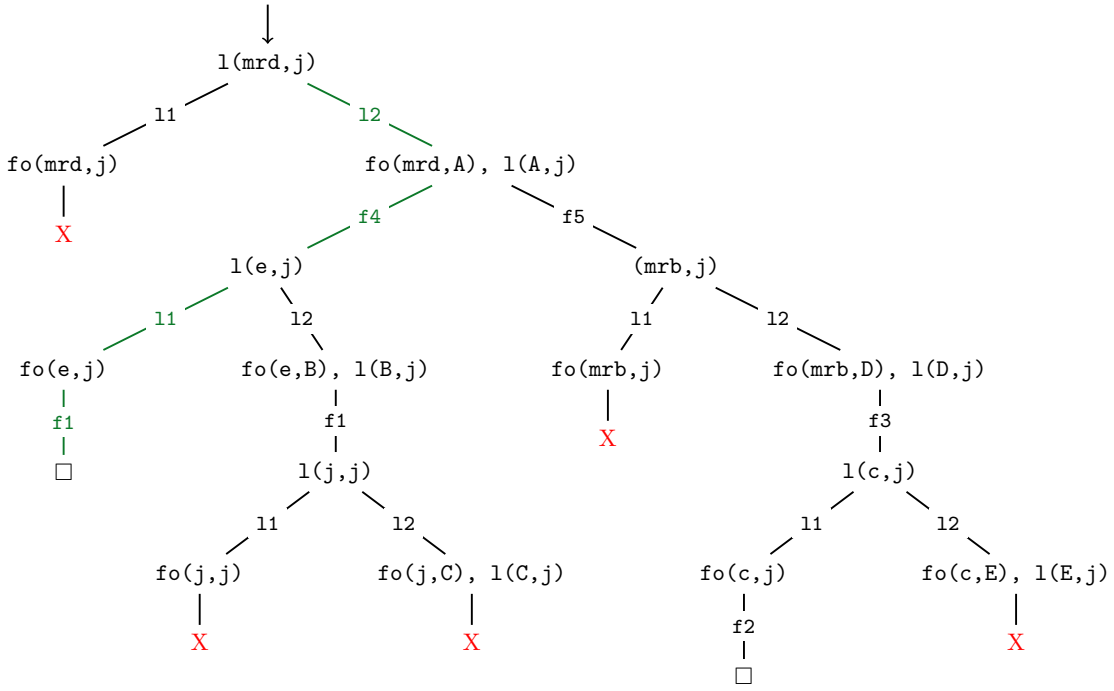


Figure 1: SLD-tree for the query `?- likes(mrdarcy, jane).`, with abbreviations `l` for `likes`, `f` for `friendof`, and so on. Branches are labelled with the rule or fact that is being applied there (corresponding to the labels in Program 2), with the first rule for `likes` that you encounter when reading the program from top to bottom is labelled `l1`, and so on. Upper case letters represent variables. An X represents a failed proof of `likes(mrdarcy, jane)`. If a *path* from the *root* of the tree to a *leaf* of the tree ends in a □, this means that that path corresponds to a proof of `likes(mrdarcy,jane)` One of the proofs for `likes(mrdarcy,jane)` is highlighted in green. Dummy variables `A` – `E` are introduced by Prolog whenever this is needed in order to find a match for a term.

However: `friendof(mrdarcy,elisabeth)` and `friendof(mrdarcy,mrbingly)` do not match, because they are not the same and do not contain any variables that can be instantiated such that the resulting terms are equal. Obviously, `friendof(mrdarcy,elisabeth)` does not match `elisabeth`.

In order to match the term t_1 = `friendof(mrdarcy,X)` with t_2 = `friendof(mrdarcy,elisabeth)`, Prolog needs to find a substitution θ such that `X` is instantiated (or *ground*) to `elisabeth` and $t_1\theta = t_2\theta$.

In order find a match in the knowledge base for the term `likes(mrdarcy,jane)`, Prolog takes the term in the query as its *goal* and searches the logic program from top to bottom until it finds a rule whose head *matches* the term. It this case, there is no such term in the knowledge base, but there might be if Prolog can find instantiations for `X` and `Y` such that it can match the term `likes(mrdarcy,jane)` with the term `likes(X,Y)`, which does occur as the head of a rule in the knowledge base.

Prolog now uses a process called *Selective Linear Definite clause resolution* (or *SLD-resolution*) to find proofs that justify substituting `X` for `mrdarcy` and `Y` for `jane`, to make at least one of the rules `l1` and `l2` in Program 2 be true. The process of SLD-resolution can be visualised in an SLD-tree, an example of which is shown in Figure 1.

Because there are two rules with `likes(X,Y)` as their heads, Prolog now has two chances of finding a proof for `likes(mrdarcy,jane)`, and thus introduces two new goals: `friendof(mrdarcy,jane)` and `friendof(mrdarcy,A), likes(A,jane)` (the bodies of the two rules), with `A` a dummy variable used in the SLD-resolution process. This is shown in Figure 1as the root splitting in two child nodes, one for each new goal. Prolog will continue recursively and again search the knowledge base looking for heads of rules that can be unified with the terms in the goals.

Take for example the first new goal: `friendof(mrdarcy,jane)` (the left child of the root in Figure 1). Prolog searches the knowledge base for a fact or rule that unifies with this term. We see that the knowledge base only includes facts that have the `friendof` predicate in their heads. As none of the facts can unify with `friendof(mrdarcy,jane)`, this line of inquiry fails and we reach a dead end. This is marked in Figure 1 by X.

Prolog can now continue searching for proofs with the other new goal (branching to the right child of the root in Figure 1): `friendof(mrdarcy,A), likes(A,j)`. Prolog first tries to unify `friendof(mrdarcy,A)` with a rule head in the database, and finds it has two options: instantiating `A` with `elisabeth` will yield a true fact from the knowledge base, and instantiation `A` with `mrbingly` also. However, the goal consisted of two terms, so the second one has to be matched with a head in a rule in the knowledge base also. This means Prolog has to try to unify `likes(elisabeth,jane)` and `likes(mrbingly,jane)` for these instantiations, respectively.

As we see in Figure 1, Prolog can find a rule (`l1`) in the knowledge base, whose head can be matched with `likes(elisabeth,jane)`. Prolog can unify the body of that rule with a fact (`fo1`), which means a proof is found for `likes(mrdarcy,jane)`. This proof is highlighted in Figure 1. Now Prolog will return `:- yes.` as an answer to the query.

For Prolog it is only relevant whether or not *a* proof can be found for the query, so normally the search would stop once one proof is found. We include however the entire SLD-tree to show that there are two proofs for this particular query.

The observant reader may have noticed that during the proof search described above, rules are evaluated in a top-to-bottom fashion. New goals found in the bodies of those rules are evaluated from left to right. This property of Prolog and its consequences for the procedural meanings of Prolog programs is explained in more detail in chapters 2 and 3 of *Learn Prolog Now!* [4].

3.2 ProbLog

The probabilistic extension of Prolog, *ProbLog*, was introduced in 2007 in *ProbLog: A Probabilistic Prolog and its Application in Link Discovery*, by Luc De Raedt et. al. [13]. This chapter is based on that article. We first introduce the additional syntax and semantics for ProbLog, after which we will discuss the inference methods used to compute success probabilities for ProbLog queries.

Syntax and semantics: adding probabilities to clauses

Where Prolog is used to describe a logic program, ProbLog is used to define a distribution over logic programs, by assigning probabilities to rules and facts. The probabilities are assumed to be mutually independent, and represent the probability that the corresponding clause is true in a randomly sampled logic program. Consider the following example:

Program 3: Pride & Prejudice (probabilistic).

```
f1      0.9::friendof(elisabeth,jane).
f2      0.2::friendof(caroline,jane).
f3      0.7::friendof(mrbingly,caroline).
f4      0.6::friendof(mrdarcy,elisabeth).
f5      0.8::friendof(mrdarcy,mrbingly).
l1      1.0::likes(X,Y) :- friendof(X,Y).
l2      0.8::likes(X,Y) :- friendof(X,Z), likes(Z,Y).
```

In the probabilistic logic program above, we have added probabilities to the facts and rules.[2] If we would pose this query to a Prolog program (such as the one in Program 5), it would return `:- yes.` if it was successful in finding a proof for `likes(mrdarcy,jane).` Posing such a query to a ProbLog program (such as the one in Program 3, however, returns the *success probability* for this query, i.e. the probability that a proof is found for `likes(mrdarcy,jane).` in a randomly sampled logic program.

More formally: suppose we have a ProbLog program $T = \{(p_1, c_1), \ldots (p_n, c_n)\}$ with p_i the probability of clause c_i. This ProbLog program defines a probability distribution over logic programs $L \subseteq L_T = \{c_1, \ldots, c_n\}$:

$$P(L \mid T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i), \tag{1}$$

where $P(L \mid T)$ represents the probability that a particular logic program L is randomly sampled from a probabilistic logic program T. A randomly sampled program can be seen as a selection of ground rules and facts from the probabilistic program, where each probabilistic rule or fact corresponds to the *atomic*

[2] If there is no probability associated with a clause, the clause effectively has probability 1.

choice of including it in the sampled program or not, based on chance. From the program above, the following ground (probabilistic) rules can be extracted:

Program 4: Pride & Prejudice (ground).

```
f1      0.9::friendof(elisabeth,jane).
f2      0.2::friendof(caroline,jane).
f3      0.7::friendof(mrbingly,caroline).
f4      0.6::friendof(mrdarcy,elisabeth).
f5      0.8::friendof(mrdarcy,mrbingly).
l11     likes(elisabeth,jane) :- friendof(elisabeth,jane).
l12     likes(caroline,jane) :- friendof(caroline,jane).
l13     likes(mrbingly,caroline) :- friendof(mrbingly,caroline).
l14     likes(mrdarcy,elisabeth) :- friendof(mrdarcy,elisabeth).
l15     likes(mrdarcy,mrbingly) :- friendof(mrdarcy,mrbingly).
l21     0.8::likes(mrbingly,jane) :- friendof(mrbingly,caroline),
                                     likes(caroline,jane).
l22     0.8::likes(mrdarcy,jane) :- friendof(mrdarcy,elisabeth),
                                    likes(elisabeth,jane).
l23     0.8::likes(mrdarcy,caroline) :- friendof(mrdarcy,mrbingly),
                                        likes(mrbingly,caroline).
l24     0.8::likes(mrdarcy,jane) :- friendof(mrdarcy,mrbingly),
                                    likes(mrbingly,jane).
```

This program contains nine atomic choices: rules `l11` – `l15` are deterministic. An example of a randomly sampled program is the following:

Program 5: A randomly sampled Pride & Prejudice program.

```
f1      friendof(elisabeth,jane).
f4      friendof(mrdarcy,elisabeth).
f5      friendof(mrdarcy,mrbingly).
l1      likes(X,Y) :- friendof(X,Y).
l21     likes(mrbingly,jane) :- friendof(mrbingly,caroline), likes(caroline,jane).
l23     likes(mrdarcy,jane) :- friendof(mrdarcy,mrbingly),  likes(mrbingly,jane).
```

where rules `l11` – `l15` are again summarised in rule `l1`. By Equation (1), the probability for this particular logic program L is:

$$P\left(L \mid T\right) = \left(.9 \cdot .6 \cdot .8 \cdot (.8)^2\right) \cdot ((1-.2) \cdot (1-.7) \cdot (1-.8)) \approx .013$$

Now we can define the success probability $P(q \mid T)$ of a query q given a probabilistic logic program T as follows:

$$P\left(q \mid L\right) = \begin{cases} 1 & \exists\theta : L \models q\theta \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$P\left(q, L \mid T\right) = P\left(q \mid L\right) \cdot P\left(L \mid T\right) \tag{3}$$

$$P\left(q \mid T\right) = \sum_{L \subseteq L_T} P\left(q, L \mid T\right), \tag{4}$$

where $P(q \mid L)$ represents the probability that a proof for q can be found (through SLD-resolution) in logic program L, $P(q, L \mid T)$ represents the probability that this logic program L is obtained through a random sampling from probabilistic logic program T *and* a proof for q exists in that logic program, and finally $P(q \mid T)$ represents the total success probability for query q, given probabilistic logic program T.

Looking at Equation (4), we see that in order to compute the success probability of a query q, we need to list all the logic programs that could possibly be obtained by randomly sampling probabilistic logic program T. As the number of possible logic programs scales exponentially with the number of probabilistic clauses in the probabilistic logic program, so does the complexity of computing the success probability of a query, making this method of computing success probabilities infeasible for larger programs. In the next chapter we describe a method for computing the success probability in a more feasible manner.

Inference: Boolean formulas and arithmetic decision diagrams

A key insight for a more feasible method of computing the success probability of a query is the fact that we need only consider those clauses that are actually involved in proofs for the query. We can identify those clauses by *grounding* the ProbLog program: performing SLD-resolution to identify all the possible proofs for the query, and the clauses that need to be true in order for the proofs to be completed.

Consider the SLD-tree in Figure 1. Two paths from root to leaf correspond to proofs for the query `likes(mrdarcy,jane).` We can associate a Boolean variable with each clause that is use on the branches along the paths that lead to a successful proof. For example, we can associate the Boolean variables ℓ_2, f_4, ℓ_1 and f_1 with the labels on the branches of the proof that is highlighted in green in Figure 1. In order for the proof to succeed, each of these variables should have value $\top$. Therefore the probability that this proof succeeds is equal to the probability that the variables ℓ_2, f_4, ℓ_1 and f_1 are all true. With a similar argument for the other proof shown in Figure 1, the total probability that the query `likes(mrdarcy,jane)` succeeds, given the ProbLog Program 3 is given by:

$$P\left(\texttt{likes(mrdarcy,jane)} \mid T\right) = P\left((\ell_{22} \wedge f_4 \wedge \ell_{11} \wedge f_1) \vee (\ell_{24} \wedge f_5 \wedge \ell_{21} \wedge f_3 \wedge \ell_{12} \wedge f_2)\right). \tag{5}$$

Here, we have converted the query to a Boolean DNF formula. Note that, as $P(\ell_{11}) = P(\ell_{12}) = 1.0$, the probability above is equal to:

$$P\left(\texttt{likes(mrdarcy,jane)} \mid T\right) = P\left((\ell_{22} \wedge f_4 \wedge f_1) \vee (\ell_{24} \wedge f_5 \wedge \ell_{21} \wedge f_3 \wedge f_2)\right). \tag{6}$$

More generally, we can write:

$$P\left(q \mid T\right) = P\left(\bigvee_{b \in pr(q)} \bigwedge_{b_i \in cl(b)} b_i\right), \tag{7}$$

with $pr(q)$ the set of proofs of a query q, and $cl(b)$ the clauses that are used in a particular proof b. Note that in practice, the ProbLog implementation does not convert queries to DNFs, but rather to the more general rooted *AND/OR DAGs*[3]. In these DAGs, internal nodes represent either conjunctions or disjunctions of their children, and 'leafs' represent variables. An example of such a DAG is shown in Figure 2.

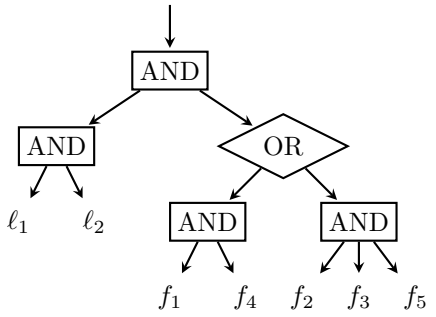


Figure 2: An example of an AND/OR (AND/OR) Directed Acyclic Graph (DAG) representing a logic formula for the existence of at least one of the two proofs shown in Figure 1. Nodes in the DAG can have an arbitrary number of parents, and an arbitrary number of children.

Now that the query is transformed, by means of grounding the probabilistic logic program through SLD-resolution, into a logic formula, the probability for that formula to evaluate to $\top$ can be computed.

A method for computing this probability is that of *Weighted Model Count (WMC)* [17]. The task of *model counting* is finding the number of models of a logic formula, i.e. the number of interpretations that cause the formula to evaluate to $\top$. WMC is the generalisation of this process, where each model is assigned a weight. By interpreting the probabilities of the variables as their weights, the weight of an interpretation (possible world) $\mathscr{I}$ of T for a given formula q can simply be defined as follows:

$$\omega(q \mid \mathscr{I}) = \begin{cases} \prod_{v_i \in \mathscr{I}} p_i \prod_{v_i \notin \mathscr{I}} (1 - p_i) & \text{if } \mathscr{I} \models q \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

with $\mathscr{I}$ the set of variables that have value $\top$ and p_i the probability of variable $v_i \in T$. The total weight of a formula q then corresponds to the probability of that formula, given the probabilistic logic program T:

$$P(q \mid T) = \sum_{\mathscr{I}} \omega(q \mid \mathscr{I}). \tag{9}$$

Note that there are 2^{N_v} possible interpretations (with N_v the number of variables in T that are relevant for query q), and the computation of the weight of each interpretation involves checking if it is a model

[3]In the rest of this work, we will simply use 'DAG' instead of 'AND/OR DAG' for shortness.

of q and, if this is the case, performing $|\mathcal{V}|$ multiplications. In the general case, WMC is #P-complete [7, 28], as it involves a disjoint sum problem. A more efficient approach is to use knowledge compilation (see also Appendix B). The aim of knowledge compilation is to convert formulas into a more compact representation. In Problog, the logic formulas that represent queries are converted into *ordered binary decision diagrams*.

An OBDD of the DNF in Equation (6) is shown in Figure 3. For an introduction in OBDDs, see Appendix B.2. Even though the implementation of ProbLog that we use for this work uses Sentential Decision Diagrams (SDDs) rather than OBDDs, we use OBDDs to explain the principles of ProbLog, as they are an easier to read, but less compact alternative to SDDs. For a description of SDDs, see Appendix B.3. In either case, the DAG representing a logic formula is used to compile the OBDD/SDD: the DAG is traversed in a bottom-up fashion, and increasingly complex OBDDs/SDDs are compiled to represent the various subformulas represented by the sub-DAGs rooted at the different nodes in the DAG. For instance, a formula $f = (\ell_{22} \wedge f_4 \wedge f_1) \vee (\ell_{24} \wedge f_5 \wedge \ell_{21} \wedge f_3 \wedge f_2)$ is converted into an OBDD by applying a disjoin operation on previously constructed OBDDs for $g = \ell_{22} \wedge f_4 \wedge f_1$ and $h = \ell_{24} \wedge f_5 \wedge \ell_{21} \wedge f_3 \wedge f_2$. Generally, OBDDs are built incrementally, by traversing the DAG in a bottom up fashion and compiling increasingly complex OBDDs by applying disjoin and conjoin operations on less complex ones. This process is similar to $T_{\mathcal{P}}$-compilation [39].

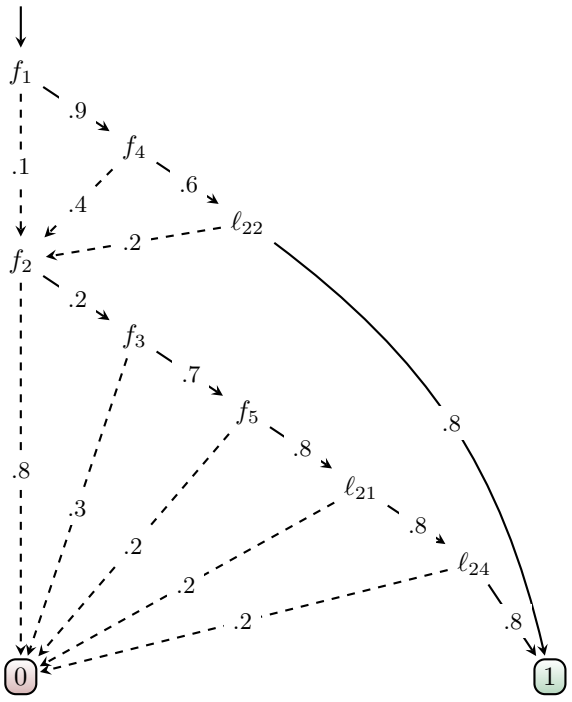


Figure 3: An Ordered Binary Decision Diagram (OBDD) for the logic formula of Equation (6) (corresponding to query `likes(mrdarcy,jane)` in Program 3). The internal nodes are labelled with the Boolean variables, their outgoing solid high (dashed low) branch with the probability that they have value $\top$ ($\bot$) in a randomly sampled probabilistic logic program. The leafs show the probability that the query evaluates to true in the interpretation of the Boolean formula that corresponds to the truth values assigned to the Boolean variables on the path from root to leaf.

Note that the OBDD in Figure 3 is different from the ones discussed in Appendix B.2 in that the leafs now represent probabilities rather than truth values, and the edges are labelled with probabilities corresponding to the probabilities in Program 3. We can compute the success probability for the query `likes(mrdarcy,jane)` in a bottom-up fashion with the algorithm shown in Algorithm 1.

Note that in Algorithm 1 p_n corresponds to the probability that clause n is true in the sampled program, while $P(h)$ ($P(\ell)$) corresponds to the output of PROBABILITY(h) (PROBABILITY(ℓ)), with h (ℓ) the high (low) child of n in the OBDD.

For our example, applying Algorithm 1 to the OBDD in Figure 3 yields a success probability

$$P(\texttt{likes(mrdarcy,jane)} \mid T) \approx 0.44.$$

In summary: by grounding the probabilistic logic program for each query using SLD-resolution, we can obtain Boolean formulas that represent the query and whose probability can be computed through OBDDs that are labelled with probabilities rather than truth values.

Note that for the evaluation of ProbLog queries, approximate and anytime algorithms have also been developed [13, 39].

3.3 DTProbLog

The decision theoretic extension of ProbLog was introduced in 2010 by Guy Van den Broeck et. al. in *DTPROBLOG: A Decision-Theoretic Probabilistic Prolog* [35]. This chapter is based on that work. We

Algorithm 1 Probability of a query.

Input: OBDD node n
Output: Success probability of query corresponding to the part of the DNF that is made up of the variable represented by n and those below it in the OBDD

1: **procedure** PROBABILITY(n)
2: **if** n is 0-terminal **then return** 0 **end if**
3: **if** n is 1-terminal **then return** 1 **end if**
4: $P(h) \leftarrow$ PROBABILITY(h) ▷ h is high child
5: $P(\ell) \leftarrow$ PROBABILITY(ℓ) ▷ ℓ is low child
6: **return** $p_n \cdot P(h) + (1 - p_n) \cdot P(\ell)$
7: **end procedure**

will start with an example program to introduce DTProbLog's syntax and semantics, and then introduce the inference methods for DTProbLog problems.

Syntax and semantics: adding decision variables and utilities

DTProbLog allows a user to model decision problems with probabilistic components, and to evaluate the utility (or expected reward) of a set of actions (decisions). Consider the classic example [35]:

Program 6: Umbrella.

```
?::umbrella.
?::raincoat.

0.3::rainy.
0.5::windy.

broken_umbrella :- umbrella, rainy, windy.
dry :- rainy, umbrella, not(broken_umbrella).
dry :- rainy, raincoat.
dry :- not(rainy).

utility(broken_umbrella, -40).
utility(raincoat, -20).
utility(umbrella, -2).
utility(dry, 60).
```

The *decision theoretic probabilistic logic program* $\mathcal{DT}$ above contains besides (probabilistic) facts and rules also *decision clauses* (the lines prefixed with `?::`) and *utilities*, or *rewards*. The program represents a decision problem about what to bring when we go outside: an umbrella, a raincoat, both or neither. The decision has to be made taking into considerations a weather forecast predicting rain and wind with certain chances, rules about consequences of choices and weather conditions, and finally reward (that may be negative) for certain choices and outcomes.

Instead of querying this program to learn about the success (probability) of a query, the program will compute the strategy $\sigma = \{\texttt{umbrella} \mapsto b_u, \texttt{raincoat} \mapsto b_r\}$, with $b_u, b_r \in \{\top, \bot\}$ (assignment of truth values to decision clauses) that yields the largest expected utility.

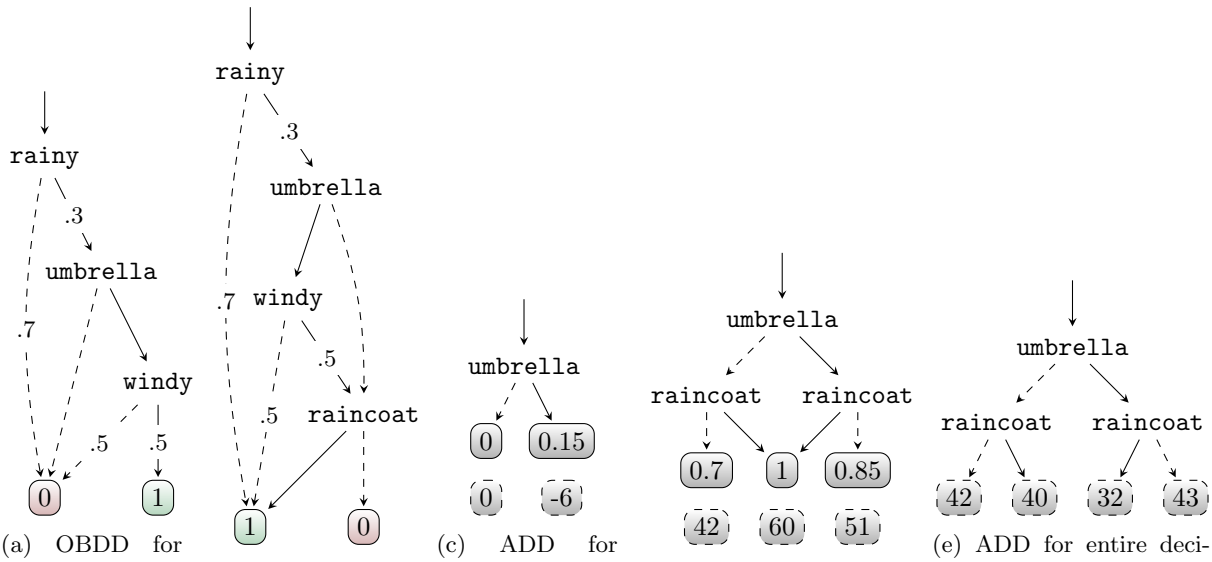
Inference: solving decision problems

Program 6 contains two decision variables: `umbrella` and `raincoat`. Solving the decision problem involves computing the total reward for the different possible assignments to decision variables (strategies). From the utilities listed at the end of the program, the utilities for just the choices `umbrella` and `raincoat` can be immediately read off, because `umbrella` and `raincoat` are chosen to be either true or not. The expected values for the other utilities have to be computed, as they involve consequences described by a probabilistic logic program.

Computing the expected values of the utilities of `broken_umbrella` and `dry` is a very natural extension of the computation of the success probabilities of queries. After all, if the success probability of the query `?- broken_umbrella.` and `?- dry.` are known, the expected values for their utilities are easily obtained by multiplying them with their respective utilities.

The first step in computing the utilities is therefore to use SLD-resolution to get Disjunctive Normal Form (DNF) formulas for `broken_umbrella` and `dry`, and using those to create OBDDs that express the interpretations of the probabilistic and decision variables in the program that make these queries return `:- true.` These OBDDs are shown in Figures 4a and 4b. The outgoing branches of the probabilistic variables are labelled with the probabilities as specified in Program 6. Success probabilities for the different strategies are computed by filling in the appropriate weights on the outgoing branches of the internal nodes representing decision variables in the diagram. For example: if the umbrella is chosen to be taken, a 1 is placed on the positive branch and a 0 on the negative branch of `umbrella` in Figure 4a. This yields a probability of 0.15 for `broken_umbrella` (computed as described in Chapter 3.2).

The next step is to compile an Algebraic Decision Diagram (ADD) [1] that represents the success probability for each query, as a function of the strategies. These are shown in Figures 4c and 4d. Note that the solid leaves show the probabilities for the strategies encoded by the paths from the root to those leaves, as they were computed using the OBDDs of Figures 4a and 4b. Note also that, since the value of `raincoat` is irrelevant to query `broken_umbrella`, there are only two strategies for this query (`umbrella` and `not(umbrella)`, while there are four different strategies for the `dry` query.



(a) OBDD for `broken_umbrella`. (b) OBDD for `dry`. (c) ADD for `broken_umbrella`. (d) ADD for `dry`. (e) ADD for entire decision problem.

Figure 4: OBDDs and ADDs used in the computation of the best strategy σ for the problem in Program 6. The solid leaves in Figures 4c and 4d represent the probabilities for `broken_umbrella` and `dry`, computed with the OBDDs in Figures 4a and 4b by 'plugging in' the different strategies that are encoded by the choices corresponding to the paths from root to leaf. The dashed leaves represent the corresponding expected values for the utilities for these outcomes.

The final step is to combine the diagrams of Figures 4c and 4d and the utilities for `umbrella` and `raincoat` to one ADD from which we can read off what the best strategy is. This ADD is shown in Figure 4e and shows that the best strategy is to bring an umbrella, but leave the raincoat at home.

In practice the ADDs shown in Figures 4c to 4e are not computed explicitly, but the outcomes are generated in an algorithm that exhaustively enumerates all possible strategies, computes the corresponding WMCs using the OBDDs and evaluates the utilities one by one.

3.4 Constraint programming and constraint satisfaction

The subject of this work is combining PLP with CP. This is a natural choice, since CP is a programming paradigm that specifies a program as a set of relations between variables in the form of constraints. This

relational approach is conceptually very similar to the declarative nature of logic programming. In fact, logic programs are a special case of Constraint Logic Programming (CLP), where solutions to problems consists of the most specific set of constraints on variables that can be derived from the logical KB [29]. In this work we focus mainly on the probabilistic logic part of this marriage between PLP and CP, but we do give a brief introduction to CP in this chapter.

The field of CP has produced a myriad of techniques for solving CSPs and COPs, which are applicable to COPs in general, and are implemented in modern day solvers [27]. In this work we focus on specific types of COPs (see Chapter 4), developing methods and heuristics that are specific to those problems. However, general solving techniques are also still applicable. Although we do not use them or implement them in this work, we introduce them briefly in this chapter, as they are relevant for a possible next step: integrating specific solving techniques presented in this work with the existing generic solver technology. This chapter is based on the *Handbook of Constraint Programming* [27].

Formal definition of constrained satisfaction problems

In the *Handbook of Constraint Programming* ([27, ch. 2]), a CSP is formally defined as a triple $\mathcal{P} = \langle X, D, C \rangle$, with $X = \langle x_1, \ldots, x_n \rangle$ an n-tuple of variables over a corresponding n-tuple of domains $D = \langle D_1, \ldots, D_n \rangle$, such that $x_i \in D_i$. The constraints are defined by t-tuple $C = \langle C_1, \ldots, C_t \rangle$, where C_j is a tuple $\langle R_{S_j}, S_j \rangle$, with R_{S_j} a relation on the variables in S_j, which represents the scope of constraint C_j. The solution to a CSP $\mathcal{P}$ is an n-tuple $A = \langle a_1, \ldots, a_n \rangle$ with $a_i \in D_i$, where C_j is satisfied for all j, meaning that R_{S_j} holds under the projection of A on C_j's scope S_j.

For solving a COP, it is not sufficient to find *some* solution, but the task is to optimise the solution with respect to a certain cost function. The goal is to find the solution with the lowest cost. Note that this *minimisation* problem can be turned into a *maximisation* problem by negating the cost function. That the search for a solution with a lower value for the cost function (i.e. a better solution) can be guided by imposing a new constraint: the value of the cost function must be lower than the best value found so far during the search process.

We now discuss the most common search strategy applied in CP, as well as some inference techniques that are used to make the search for an (optimal) solution more efficient.

Solving techniques

Most solvers for CSPs and COPs use a combination of *search* and *inference* techniques, which we will briefly discuss here.

The search part of the solving process focusses on traversing the space of possible solutions in a structured manner. The most commonly used search technique is that of *backtracking*: a Depth-First Search (DFS) traversal of a search tree. This work is limited to binary domains of the variables in the COPs, therefore the corresponding search trees are binary trees. The root of the search tree corresponds to an empty set of assignments to variables: $C = \varnothing$. Branching on this root node corresponds to choosing one variable to have a value of either true ($\top$) or false ($\bot$). This choice is added to C, and branching continues in each of the children of the root in turn, in a depth-first manner. In each node, the constraints are checked to see if they are still satisfied under the partial assignment to variables that corresponds to the node. If at least one constraint is violated, the current branch is a dead end, and backtracking is needed by undoing the latest assignments to variables (in the reverse order of the one in which they were made), until a variable is found for which there remains a value in the domain that has not yet been explored. This type of backtracking forms the basis for the new algorithms that are proposed in Chapters 6.3 and 6.4.

The efficiency of any backtracking technique can be improved by inference techniques such as *constraint propagation*. This technique uses constraints in nodes of the search tree in order to rule out *local inconsistencies*. A local inconsistency is an instantiation of some of the variables that in itself satisfies all constraints, but does not allow the addition of the assignment of any domain value to at least one of the unassigned variables.

Consider for example a node colouring problem on a path graph with four nodes $\{t, u, v, w\}$ (with undirected edges (t, u), (u, v) and (v, w)) and two colours (peach and turquoise). Suppose the constraint is that adjacent nodes cannot have the same colour. Suppose that the backtracking search algorithm first assigns the colour peach to node t and turquoise to v. Even though this assignment does not violate the

constraint, any colour assignment to u will cause a constraint violation. Thus, this is an example of local inconsistency.

Suppose the backtracking algorithm now assigns a value to w and saves the assignment to v for last and only then finds out that there are no valid domain values left to choose from for this variable. For this small toy problem, not much time is wasted in this unfruitful line of search. However, for larger problems it can cause a lot of search effort to be done in vain, if it is not detected that a variable has an empty set of remaining domain values. By checking the validity of domain values for all unassigned variables ('looking ahead'), search can be pruned whenever no valid values are found for an unassigned variable.

A special case of local consistency is that of *arc consistency*. Where the looking ahead method described above only considers valid value assignments for single variables, maintaining arc consistency involves the existence of valid assignments for *pairs of variables*.

Consider an assignment C to all variables in a constraint c, with $C[x]$ the value that is assigned to variable x. A value $a \in D_x$ has a *support* in constraint c if there exists an assignment C such that $a = C[x]$ and $C[y] \in D_y$ for any variable y in c. The constraint c is said to be *arc consistent* if for each variable in c, each value in the corresponding domain has a support in c. This arc consistency is maintained by removing values from the various domains, until all constraints are either arc consistent, or all values are removed from the domain of one or more variables. All removed values correspond to branches that can be pruned during the search, since they correspond to assignments of values to variables that are locally inconsistent with the current partial assignment C.

A generalisation of arc consistency is that of *path consistency*, where not the consistency of individual variables is considered, but the consistency of tuples of variables.

These are just a few examples of search and inference techniques that are developed in the CP community, but there are many other techniques available. For example: symmetries of constraint problems can be exploited to speed up inference and search or approximate algorithms can exploit local search to quickly find very good (although maybe not perfect) solutions to COPs. Different techniques exist for different domains and different types of problems, like linear programming (or linear optimisation) for problems with linear constraints, or techniques for solving (combinatorial) problems on structured domains, and even techniques for solving problems whose precise definition is uncertain or subject to change. For an overview, see *Handbook of Constraint Programming* [27].

4 Problem statement

In this work we research optimisation methods for solving stochastic constraint optimisation problems SCOPs. We consider problems that can be described by a probabilistic logic program, containing decision variables, and that can be solved using a combination of PLP and CP techniques. Specifically, we consider problems of the following form:

Problem 1. *Given a probabilistic logic program T with a set of probabilistic clauses $\mathscr{P} \subseteq T$ and a set of decision clauses $\mathscr{D} \subseteq T$ (deterministic clauses are simply probabilistic clauses with probability 1), we consider constrained optimisation problems whose solution consists of a* strategy *σ mapping each decision variable $d \in \mathscr{D}$ to a truth value. The constraints under consideration are of a* probabilistic *nature and come in two forms:*

$$P\left(q_c \mid T, \sigma\right) \leq \vartheta \tag{10}$$

$$P\left(q_c \mid T, \sigma\right) \geq \vartheta, \tag{11}$$

with $P(q_c \mid T, \sigma)$ the success probability *of a query q_c (see Chapter 3.2), given probabilistic logic program T and a strategy σ, and $0 \leq \vartheta \leq 1$. We consider* probabilistic optimisation criteria *of the forms:*

$$\max_{\sigma}\left(P\left(q_o \mid T, \sigma\right)\right) \tag{12}$$

$$\min_{\sigma}\left(P\left(q_o \mid T, \sigma\right)\right), \tag{13}$$

where $P\left(q_o \mid T, \sigma\right)$ is the success probability of query q_o, given T and σ. We also consider counting optimisation criteria *of the forms:*

$$\max_{\sigma}\left(|\sigma_{\top}|\right) \tag{14}$$

$$\min_{\sigma}\left(|\sigma_{\top}|\right), \tag{15}$$

where $|\sigma_{\top}|$ is the size of the set of decision clauses that are chosen to be true in strategy σ. Note that in this optimisation setting, there is no optimisation query for which to compute the probability, only constraint queries. The optimisation task is to maximise (or minimise) the size of σ. This work is limited to problems with one optimisation criterion and probabilistic constraints that are all of the same form.

In this work, we focus specifically on probabilistic logic programs that specify *paths in networks*, as problems regarding gene expressions, viral marketing and viral epidemics can often be formulated in such a way, and ProbLog is typically used for these kinds of problems. We also focus on two particular settings for the optimisation strategy (the *maximisation* settings), and one particular type of probabilistic constraints (an upper bound on probabilities, i.e. Equation (11). We define this more specific problem as follows:

Problem 2. *Given a probabilistic logic program T describing a graph $G(V, E)$. The edges $(u, v) \in E = E_d \cup E_p \cup E_c$ are represented by decision variables (E_d), probabilistic variables (E_p) or constants (i.e. deterministic, E_c). Combinations are also allowed, for example: if an edge e is represented both by a probabilistic variable and a decision variable, the edge exists with a probability p_e if the corresponding decision variable $d_e = \top$, and 0 otherwise. The vertices in $V = V_p \cup V_c$ may be either probabilistic (V_p) or deterministic (V_c). Probabilistic logic program T does not contain negation.*

We select a strategy σ mapping edge decision variables $e_d \in E_d$ to truth values in order to optimise one of the following two quantities:

maxProb *Maximise the probability that there is a path from node a to node b:*

$$\max_{\sigma}\left(P\left(path(a, b) \mid T, \sigma\right)\right) \tag{16}$$

maxSet *Maximise the the size of the set of decision variables with value $\top$:*

$$\max_{\sigma}\left(|\sigma_{\top}|\right) \tag{17}$$

subject to probabilistic constraints of the form:

constraint *The probability that there is a path from node a to node c should not exceed threshold ϑ:*

$$P\left(path(a,c) \mid T, \sigma\right) \leq \vartheta, \tag{18}$$

with $0 \leq \vartheta \leq 1$

Note that the graphs described by probabilistic logic program T are *unweighted*. The edges might exist with a certain probability, but if they do exist, they have weight 1. As T contains no negation, the probability that a path exist can never decrease if a decision variable is added to the set of 'true' decision variables: the probabilities are *monotonic*. In the next chapter we present a number of examples of problems of this form.

The first steps of the naive way of solving such a probabilistic constrained optimisation problem using probabilistic logic programming are described in Chapters 3.2 and 3.3. The program is compiled into a set of logic formulas (one for each query that represents a probabilistic optimisation criterion or constraint), represented by an *AND/OR logic directed acyclic graph* (DAG, see Appendix B.1) for each formula. Using the DAG representations of these formulas, they are compiled in a bottom-up fashion into sentential decision diagrams (SDD, see Chapter 3.2 and Appendix B.3). The problem is solved exhaustively by generating all possible strategies for the set of decision variables, using each strategy as *evidence* in the SDDs and computing the WMC for each of the formulas to evaluate the value of the optimisation criterion and check if the constraints are respected. The strategy that yields the largest value for the optimisation criterion is returned, along with that value.

Observe that this naive strategy requires the compilation of queries to SDDs. Darwiche provides some limited indications of the time complexity of SDDs [10], an analysis of which is outside the scope of this work. Consider therefore the empirical results on DAG and SDD sizes and compilation times shown in Figure 5.[4]

The figure shows benchmark results for the compilation of DAGs and SDDs for one or five queries on programs with 47 to 139 variables. Observe that SDDs grow more rapidly with the number of variables than DAGs (with one to four orders of magnitude), and that the compilation time for the largest SDD is five orders of magnitude larger than that for the largest DAG. We also see that SDD managers for problems with five queries are one to five orders of magnitude larger than SDD managers for problems with only one query.

Experiments for larger problems were terminated: while the compilation of their DAGs took (tenths of) seconds, the compilation of the corresponding SDDs was not finished after 48 hours.

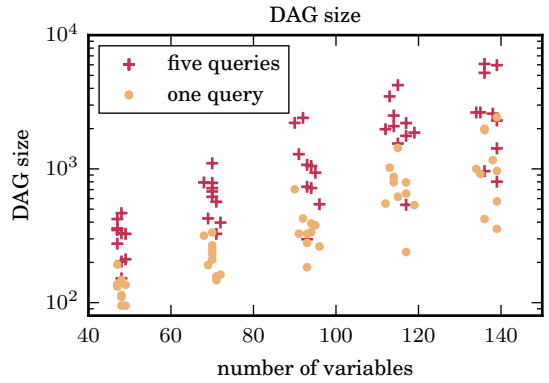
Clearly, compiling SDDs for larger problems becomes infeasible, even if the compilation of DAGs is still feasible. This observation is one of the motivations for this work.

Observe that, while we need to find an exact value for the optimisation criterion, we do not necessarily need exact values for the constraints for each strategy. As long as we are certain that the constraints are either violated or respected, bounds on the constraint are sufficient. This means that compiling the SDDs for the constraints may not be necessary. The computation of the WMC given an SDD and an strategy for the set of decision variables can be done efficiently. However, the compilation of SDDs is a bottleneck for the exhaustive approach described above for all but the smallest problems, as we have shown above. Therefore the key focus of this work is to evaluate whether strategies that avoid compiling SDDs are feasible. This focus yields the following key questions we seek to answer in this work:

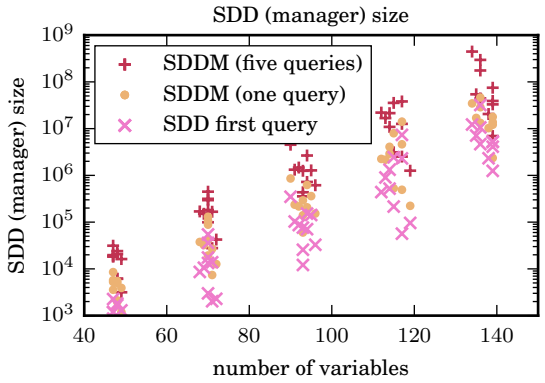
1. What heuristics and search strategies can we employ to avoid the compilation of large SDDs?
2. How effective are these methods?
3. Which properties of the SCOPs under consideration explain this effectiveness?

In the next chapter we describe artificially generated example problems used in our evaluation of SCOPs solving techniques.

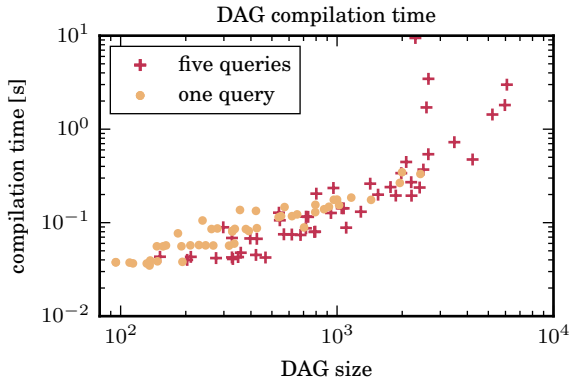
[4]For details on the used example problems, see Chapter 5. Details about the experimental setup are presented in Chapter 7.



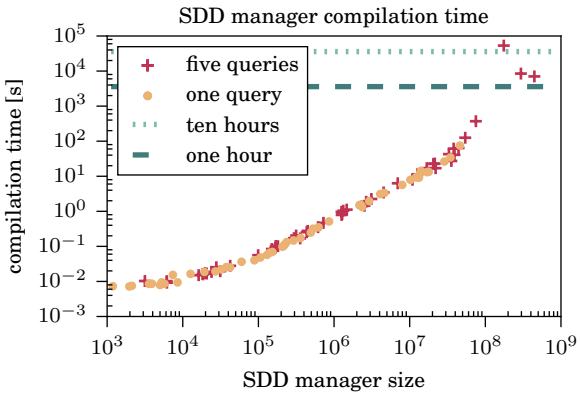
(a) Size of the DAG (see Appendix B) as a function of the number of variables (lin-log scale).



(b) Size of the SDD manager and of the SDD representing the first query (see Appendix B) as a function of the number of variables (lin-log scale).



(c) Compilation time of DAGs as function of its size (log-log scale).



(d) Compilation time of SDD manager as function of its size (log-log scale).

Figure 5: Compilation benchmark for DAGs and SDDs on `fas` dataset (see Chapter 5), for problems with one or five formulas/queries. For more details on the experimental settings, see Chapter 7.

5 Instances of Stochastic Constraint Programming

In this chapter we describe the examples of instances of probabilistic constraint programming used for this work. All example problems contain probabilistic variables and decision variables, and we can define probabilistic constraints on each of them.

We use three sets of artificially generated examples of probabilistic constraint programming: `messages`, `tell-your-friends` and `friends-and-smokers`. We describe the logic programs that define the example problems in this chapter. Details about how exactly the underlying data for these programs were generated, can be found in Chapter 7.1.

Messages

The examples in the `messages` (`mes`) problem set each represent a social network like the criminal communication network described in Chapter 1. The nodes represent people and each edge is represented by a decision variable and a probabilistic variable. If we allow an edge (u, v) to exist (by choosing a strategy σ such that the decision variable `d(u,v)` has value $v_\sigma(\texttt{d(u,v)}) = \top$), the corresponding probabilistic variable `c(u,v)` determines the probability that, if `u` receives a message, they will pass it on to `v`. Examples in this problem set encode probabilistic graphs in which we can query the probability that a message can travel from one person (the source) to another (a target), where we choose which connections between individuals are allowed to exist with a certain probability (we call these connections *active*), and which are not (*inactive*). An example of a `mes` problem instance is shown in Figure 6.

```
?::d(a,b).
?::d(a,c).
?::d(b,a).
?::d(b,c).
?::d(c,a).
?::d(c,b).

0.4::edge(a,b).
0.5::edge(a,c).
0.1::edge(b,a).
0.8::edge(b,c).
0.1::edge(c,a).
0.7::edge(c,b).

connection(X,Y) :- d(X,Y), edge(X,Y).
message(X,Y) :- connection(X,Y).
message(X,Y) :- connection(X,Z), message(Z,Y).

query(message(a,b)).
query(message(a,c)).
```

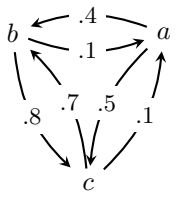


Figure 6: An example of a `mes` decision theoretic probabilistic logic program. Edges are represented by both decision variables (`d()`) and probabilistic variables (`edge()`). Nodes exist if they are the begin- or endpoint of connections that exist, so they are represented only implicitly.

In this case, we query the probability that a message sent from a certain source reaches a certain target. A **maxProb**-type optimisation criterion an example in the `mes` problem sight might be: *maximise the probability that a message sent by a reaches b.* A **maxSet**-type optimisation criterion for such a problem might be: *maximise the number of active edges.* A constraint might be: *the probability that a message sent by a reaches c cannot exceed 0.7.*

Note that for this type of problem there are exactly as many decision variables as probabilistic variables.

Note also that the nodes are encoded implicitly rather than explicitly, as they only exist if they happen to be the begin point or endpoint of an edge.

Tell-your-friends

The `tell-your-friends` (`tyf`) set of example problems is similar to the `mes` problem set in that it is about messages being sent through edges by people (nodes). Edges are represented *either* by decision variables, *or* by probabilistic variables. A node is selected as the source node. All its outgoing edges are decision variables and it has no incoming edges. All other edges are probabilistic. Except for the outgoing edges of the source, there is an edge (v, u) for each edge (u, v). In these example problems the source has a piece of news that they want to share with their friends. The outgoing edges (decision variables) represent the source either telling the friend at the receiving end of the corresponding edge the bit of news, or not telling them. The message is then sent onwards with a probability that is on the edges. An example is shown in Figure 7.

```
?::d(me,a).
?::d(me,b).
?::d(me,c).

0.8::edge(a,them).
0.8::edge(b,them).
0.4::edge(b,d).
0.6::edge(c,d).
0.3::edge(c,e).
0.2::edge(d,b).
0.3::edge(d,c).
0.7::edge(d,you).
0.3::edge(e,c).
0.9::edge(e,you).
0.4::edge(them,a).
0.5::edge(them,b).
0.8::edge(you,d).
0.1::edge(you,c).

connection(X,Y) :- d(X,Y).
connection(X,Y) :- edge(X,Y).
message(X,Y) :- connection(X,Y).
message(X,Y) :- connection(X,Z), message(Z,Y).

query(message(me,you)).
query(message(me,them)).
```

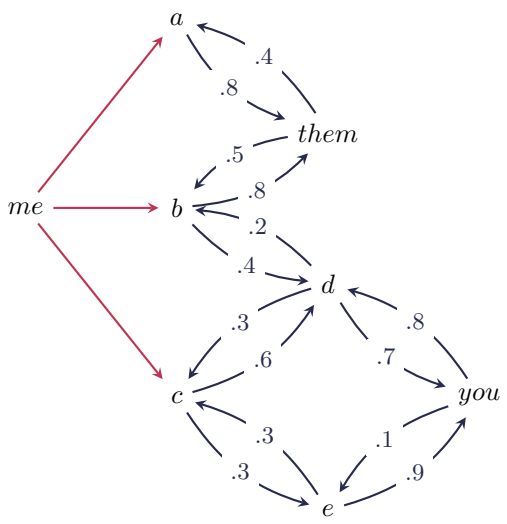


Figure 7: An example of a `mes` decision theoretic probabilistic logic program. Edges are represented *either* (`d()`) by decision variables *or* by probabilistic variables (`edge()`). Nodes are implicitly represented as begin- or endpoints of connections that exist in the program.

An example of a **maxProb**-type optimisation criterion for problems in the `tyf` problem set is: *maximise the probability that a message sent by `me` reaches `you`.* A **maxSet**-type example is: *maximise the number of friends that I call to tell them my bit of news.* A constraint might be: *the probability that a message sent by `me` reaches `them` can not exceed 0.5.*

A difference between the `mes`-type problems and the `tyf` problems is that for the latter, we need not

assign a truth value of $\top$ to a number of decision variables in order to create *one* proof for a message from source to target.

Friends-and-smokers

The underlying program of the `friends-and-smokers` (`fas`) problem set is taken from the ProbLog tutorial by the KU Leuven DTAI team [19]. It represents a social network where nodes represent people and directed edges represent friendships. Each person is stressed with a certain probability, and being stressed causes them to smoke. Furthermore, there is a probability that a certain person `Y` influences another person `X`. If `Y` indeed influences `X`, `Y` is a friend of `X` and `Y` smokes, then `X` also smokes. An example of such a program and the underlying social network is shown in Figure 8.

```
?::friend(a,b).
?::friend(a,c).
?::friend(b,a).
?::friend(b,c).
?::friend(c,a).
?::friend(c,b).

person(a).
person(b).
person(c).

0.3::stress(X) :- person(X).
0.2::influences(X,Y) :- person(X), person(Y).
smokes(X) :- stress(X).
smokes(X) :- friend(X,Y), influences(Y,X), smokes(Y).

query(smokes(b)).
query(smokes(c)).
```

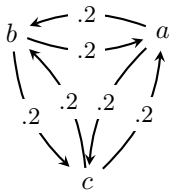


Figure 8: An example of a `fas` decision theoretic probabilistic logic program. Each edge in the graph is represented by a decision variable (`friend()`) and a probabilistic variable (`influences()`). Nodes are represented by constants (`person()`) and probabilistic variables (`stress()`).

The probabilistic logic program in the figure shows two queries: `smokes(b).` and `smokes(c).`. These can be used both as optimisation criterion (for example: *maximise the probability that person* `b` *smokes*) or as constraint (for example: *the probability that person* `c` *smokes cannot exceed 0.4*).

In this set of SCOPs, the decision variables are the friendships between the different people. The probabilistic variables come in two kinds and all have the same value. The first type represents the probability that a person is stressed. The second kind represents the probability that one person influences another. Note that a `fas` problem has n_f decision variables and $n_f + n_p$ probabilistic variables, with n_p the number of people and n_f the number of friendships. Note that each example problem is such that for each friendship `friend(u,v)` there is a friendship `friend(v,u)`.

6 Approach

In this chapter we describe our proposed method for solving COPs on probabilistic graphs. We describe the algorithms for the **maxProb** setting with one or more constraints of the $P(path(a,b)) \leq \vartheta$ type. Recall that the probabilistic logic program T describing an instance of a problem does not contain negation and the edges of the probabilistic graphs are unweighted. Thus, probabilities behave monotonically.

We start with a description of the naive methods that compile the entire program into one big SDD and use that to evaluate the different strategies. These methods are based on exhaustively enumerating all strategies (the ExhaustiveSearch method) and on performing a depth-first search (the dfs method), pruning where possible. Then we discuss some general optimisation methods that are used in the dfsapproach, as the incremental methods described next also use these optimisations.

Recall from Chapter 4 that our goal is to limit the size of the SDDs that are compiled, as compilation times are infeasibly large for all but the smallest SDDs. We describe the Incremental method in which not a big SDD is built out of both probabilistic and decision variables, but where SDDs are built incrementally, out of only the relevant probabilistic variables. Then we propose a lazy version of this incremental method (LazyIncremental), which limits the size of the built SDDs even more. We end with a short discussion on how these methods can be used for the **maxSet** optimisation setting, and why they cannot trivially be used for optimisations of the types **minProb** and **minSet**, and constraints of the type $P(path(a,b)) \geq \vartheta$. The algorithms presented in this chapter, and some of their characteristics, are summarised in Table 1.

Table 1: Some properties of the constrained optimisation problem solving algorithms discussed in this Chapter.

Algorithm	**Base strategy**	**SDD compilation**	**Variables in SDD**	**Optimisation support**
ExhaustiveSearch	enumeration	full	deterministic & probabilistic	none
dfs	DFS	full	deterministic & probabilistic	general
Incremental	DFS	incremental	probabilistic	general and incremental method specific
LazyIncremental	DFS	incremental	probabilistic	general and incremental method specific

6.1 Naive methods: building big SDDs

We compare two strategies based on the naive method of solving decision-theoretic probabilistic logic problems as described in Chapter 4. For this method, the queries in the program $\mathcal{DT}$ describing the problem instance are each compiled into a DAG, and then into an SDD. Then all different strategies are generated and added as evidence into the SDDs. Given a query q and a strategy σ, the probability $P(q \mid \sigma, \mathcal{DT})$ is computed. If the query is an optimisation criterion, the probability is compared to the best found value so far. If the query is a constraint, its probability is compared to the appropriate threshold. The best value for the optimisation criterion and its corresponding solution are updated if appropriate.

The two strategies based on this method we compare are ExhaustiveSearch and dfs. The first strategy simply generates all possible strategies one by one. Its pseudocode is given in Algorithm 2. Note that this strategy requires at least 2^{N_d} evaluations of the optimisation criterion, with $N_d = |\mathscr{D}_{\mathscr{F}}|$ the number of decision variables in $\mathcal{DT}$ that are relevant for the optimisation and constraint queries in $\mathscr{F}$. The number of constraint evaluations depends on the number of constraints, the value of their corresponding thresholds ϑ, the order in which they are evaluated and the found strategies that respect the constraints. Thus the *online* complexity (that of the search itself, excluding preprocessing) in terms of query evaluation is $\Theta\left(N_o 2^{N_d}\right)$, with N_o the number of optimisation criteria (in the scope of this work: 1), and $O\left((N_c + N_o)\, 2^{N_d}\right)$, with N_c the number of constraints. Observe that these naive methods also

require pre-compiled SDDs as input, which is very costly for all but the smallest problems, as was shown in Figure 5 in Chapter 4.

Algorithm 2 Exhaustive search. The procedure WMC is analogous to the one described in Algorithm 1, only for SDDs rather than OBDDs. Abusing notation, we use the same symbol for a (sub) SDD as for the root of that sub SDD.

Input: a compiled SDD with a set of roots of sub SDDs that encode logic formulas, representing the probabilistic constraints, each with a corresponding threshold: $(f_c, \vartheta_c) \in \mathscr{C}$. Also a root of the sub SDD representing an optimisation criterion f_o.
Output: solution to constrained optimisation problem

```
1:  procedure ExhaustiveSearch(𝒞, f_o)
2:      v_max ← 0, σ_sol ← None
3:      ℱ ← {f_c | (f_c, ϑ_c) ∈ 𝒞} ∪ {f_o}
4:      for strategies σ ∈ S_{𝒟_ℱ} do        ▷ with S_{𝒟_ℱ} the set of strategies for the decision variables in 𝒟_ℱ
5:          v_o ← WMC(f_o, σ)
6:          if v_o > v_max then
7:              c_ok ← CheckConstraints(𝒞, σ)
8:              if c_ok then v_max ← v_o, σ_sol ← σ end if
9:          end if
10:     end for
11:     return v_max, σ_sol
12: end procedure

13: procedure CheckConstraints(𝒞, σ)
14:     for (f_c, ϑ_c) ∈ 𝒞 do
15:         v_c ← WMC(f_c, σ)
16:         if v_c > ϑ_c then return False end if
17:     end for
18:     return True
19: end procedure
```

The second strategy (DFS) is based on a depth-first search. The pseudocode for this strategy, in its most naive form, is shown in Algorithm 3. With this strategy we can associate a search tree (see Figure 9 for an example). In the root of the tree, we start with no assignments to decision variables. Each internal node of the search tree corresponds to a choice to assign either $\top$ or $\bot$ to a variable (where in a node at level δ, a choice for variable $d_{\delta+1}$ is made, with the root being level 0). The leaves correspond to full strategies.

We want to be able to perform pruning whenever we find that the current partial choice of assignments to decision variables cannot yield a solution that respects the constraints. Therefore, when the algorithm enters an internal node, it first checks if the constraints are satisfied according to the partial assignment that corresponds to that node (line 15 in Algorithm 3). However, as we use the WMC method to compute probabilities, we cannot simply use that *partial* assignment, but need to provide weights (truth values) for *all* decision variables. As we can only prune whenever we are certain that the constraints cannot be satisfied, we compute a *lower bound* for each query (both constraint and optimisation) by assuming all unassigned variables to have value $\bot$.

If the constraints are indeed satisfied, we first branch on 'true' for the next decision variable (line 18), because we look at *maximisation* problems and hope to find a good solution soon. The algorithm continues recursively, until it returns to the internal node, after which it branches on 'false' (line 20).

Only in leaf nodes we evaluate the value for the optimisation criterion (line 8), as we do need an exact value for this query. If it turns out to be a better value than the best value found so far, and the current strategy respects the constraints, the best value and its corresponding strategy are updated. We evaluate the optimisation criterion first because in this work we only consider problems with one optimisation criterion, but they might have multiple constraints, so we expect it to be generally more efficient to evaluate the optimisation criterion first.

The full search tree corresponding to the DFS approach contains $\sum_{i=0}^{N_d} 2^i = 2^{N_d+1} - 1$ nodes. The evaluation of the optimisation query happens only in leafs of the tree. In a worst case scenario, the

Algorithm 3 Depth-first search with fully compiled SDDs. The CheckConstraints method is that of Algorithm 2, the WMC method is analogous to Algorithm 1. Abusing notation, we use the same symbol for a (sub) SDD as for the root of that sub SDD.

Input: a compiled SDD with a set of roots of sub SDDs that encode logic formulas, representing the probabilistic constraints, each with a corresponding threshold: $(f_c, \vartheta_c) \in \mathscr{C}$. Also a root of the sub SDD representing an optimisation criterion f_o.
Output: solution to constrained optimisation problem

```
1: procedure DFS(𝒞, f_o)
2:     ℱ ← {f_c | (f_c, ϑ_c) ∈ 𝒞} ∪ {f_o}                    ▷ Global constant
3:     return Branch(∅, 0, 𝒞, f_o, 0, ∅)
4: end procedure

5: procedure Branch(C_δ, δ, 𝒞, f_o, v_max, σ_sol)
6:     σ ← C_δ ∪ {(d ↦ ⊥) | d ∈ 𝒟_ℱ, (d ↦ b) ∉ C_δ, b ∈ 𝔹}

       Base case: all decision variables have been assigned a value
7:     if δ = N_d then
8:         v_o ← WMC(f_o, σ)
9:         if v_o > v_max then
10:            c_ok ← CheckConstraints(𝒞, σ)
11:            if c_ok then v_max ← v_o, σ_sol ← C_δ end if
12:            return v_max, σ_sol
13:        end if

       Recursive case: some decision variables remain unassigned
14:    else
15:        c_ok ← CheckConstraints(𝒞, σ)
16:        if c_ok then
               Branch on ⊤
17:            C_{δ+1} ← C_δ ∪ {(d_{δ+1} ↦ ⊤)}
18:            v_max, σ_sol ← Branch(C_{δ+1}, δ + 1, 𝒞, f_o, v_max, σ_sol)

               Branch on ⊥
19:            C_{δ+1} ← C_δ ∪ {(d_{δ+1} ↦ ⊥)}
20:            v_max, σ_sol ← Branch(C_{δ+1}, δ + 1, 𝒞, f_o, v_max, σ_sol)
21:        end if
22:    end if
23:    return v_max, σ_sol
24: end procedure
```

DFS method would therefore perform as many evaluations of the optimisation criterion as the ExhaustiveSearch method: 2^{N_d}, as this is the number of leafs in the search tree. Constraint queries are evaluated in internal nodes of the search tree, and in leafs only if the probability for the optimisation criterion exceeds the best value found so far. In a worst case, the naive DFS method performs twice as many evaluations as the ExhaustiveSearch method. Thus has a worst case complexity of $O\left(N_o 2^{N_d} + N_c 2^{N_d+1}\right)$ (and a lower bound of $\Theta(1)$). Thus, optimisations such as a sufficient amount of pruning, are needed for DFS to outperform ExhaustiveSearch.

Note that the DFS method could use an upper bound for the optimisation criterion to prune the search. It could simply assume all unassigned variables have value $\top$ and compute the probability for the optimisation query. If the resulting value is lower than the best value found so far, the search tree can be pruned. We do not yet use this pruning method, because it requires the full compilation of the SDD that represents the optimisation query. In this work, we investigate methods that avoid building full SDDs, therefore we do not yet use this type of 'upper bound' pruning, as it would not be applicable in the other algorithms proposed in this chapter.

Both the ExhaustiveSearch and dfs method stop the search once a solution with the 'perfect score' is found: 1.0 in case of the **maxProb** setting. Note that this is typically very unlikely to happen for the `mes` and `fas` problem types described in Chapter 5, although it might happen for the `tyf` problems. In addition to the 'lower bound' pruning to limit the size of the search tree, our dfs implementation makes use two optimisations that are discussed in the next chapter.

6.2 General optimisations

In this chapter we discuss a number of optimisation methods that we have implemented for DFS-based search techniques: dfs, Incremental and LazyIncremental. Note that we do not include these optimisations explicitly in the pseudocode, for reasons of simplicity and readability.

Simple subset pruning

The simple subset pruning optimisation (sssp) is based on the following principle. Given a strategy $\sigma = \{(d \mapsto b) \mid d \in \mathscr{D}_{\mathscr{F}}, b \in \mathbb{B}\}$, which maps relevant decision variables to Boolean values, and respects all constraints. Now $\sigma_\top$ ($\sigma_\bot$) represents the set of decision variables that are assigned the value $\top$ ($\bot$), such that $\sigma_\top \cap \sigma_\bot = \varnothing$ and $\sigma_\top \cup \sigma_\bot = \{d \mid (d \mapsto b) \in \sigma\}$. The strategies $\sigma' = \{(d \mapsto \top) \mid d \in \sigma'_\top\} \cup \{(d \mapsto \bot) \mid d \in \sigma'_\bot\}$, with $\sigma'_\top \subset \sigma'_\top$ and $\sigma'_\bot = \sigma_\bot \cup (\sigma_\top \cap \sigma'_\top)$, are also solutions that respect the constraints.

However, these strategies will not yield larger values for the optimisation criterion because of the monotonic behaviour of the query probabilities. We therefore need not evaluate those strategies.

We apply a simple method for pruning the search tree based on this principle, illustrated in Figure 9. Whenever we find a solution at a leaf, we prune the part of the search tree that is left of the longest path of consecutive 'true' choices (solid branches in Figure 9) and ends in the leaf. In other words: if the last 'false' choice was made at depth δ for decision variable $d_{\delta+1}$, we do not branch on 'false' for the ancestors d_i of the leaf for $i \leq \delta + 1$, and thus go back to depth $\delta - 1$ to continue branching.

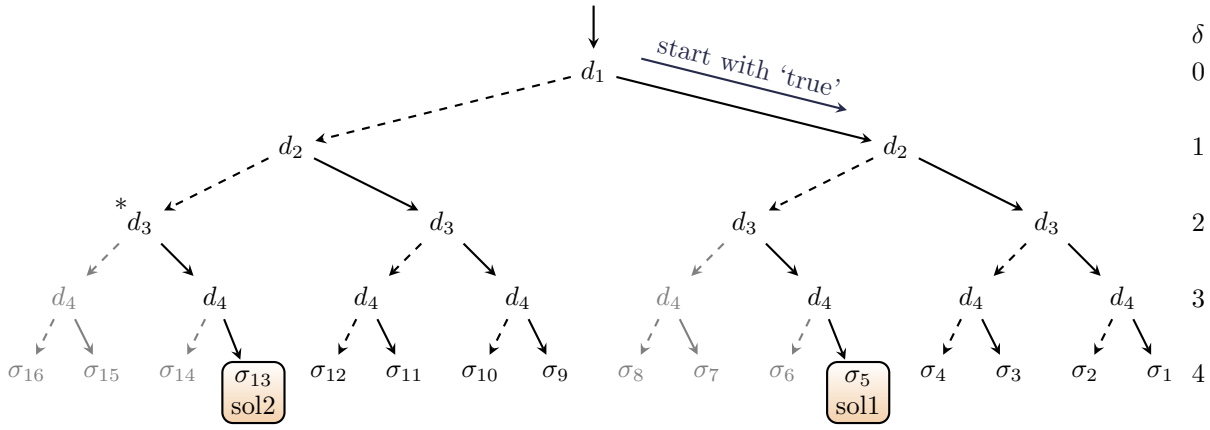


Figure 9: An illustration of subset pruning on a problem with four decision variables. The labels of the internal nodes represent the decision variable that is being branched on in that node. The labels in the leaves represent the strategies in the order in which they are encountered. In the simple version of subset pruning, the greyed-out branches are pruned once solution 1 (sol1) is found. A more sophisticated version would also prune the sub tree rooted at the d_3 node marked with $*$. Note that the tree is traversed from right to left; always starting with the 'true' branch (solid branches), and then branching on 'false' (dashed branches).

Consider the search tree in Figure 9. We see that a solution is found for strategy $\sigma_\top = \{d_1, d_3, d_4\}$, $\sigma_\bot = \{d_2\}$ (σ_5 in the figure). Now we need not continue branching to strategies σ_6 — σ_8. Similarly, we will find a solution at σ_{13} and can prune again. This optimisation works particularly well for loose constraints, as it is likely that for those constraints, solutions are found with many decision variables set to $\top$, allowing a lot of pruning if they happen to be in a long chain to a leaf.

Note that we would actually also not need to consider any of the strategies in the sub tree rooted at the d_3-node labelled with $*$, but in our implementation of simple subset pruning, we do not yet prune that entire sub tree.

Finally, we still do consider many other strategies σ' for which $\sigma'_\top \subset \sigma_\top$ holds for a found solution σ, so this simple subset pruning can still be expanded into a less simple version.

Validity check

While SSSP is an optimisation designed to limit the size of the search tree, the validity check is a method for limiting the number of probabilistic constraint checks.

It is based on the observation that when we check if the constraints may still be satisfied given the current partial assignment (line 15), we compute a *lower bound* for the probability of the constraint queries by assuming all unassigned decision variables to have value $\bot$. Therefore, if the algorithm enters a node n_δ at depth δ in the search tree through a negative branch (dashed branch in Figure 9), such that $\bot$ has just been assigned to decision variable d_δ, and we need not check the constraints again, because they were already checked for this partial assignment in n_δ's parent. We know that the current partial assignment may still yield a valid solution, and we need not check the constraints again.

Observe that this has consequences for the worst case complexity of the DFS method: because for half of the nodes in the search tree, an evaluation of the constraint queries is no longer necessary, the worst case complexity changes to $O\left((N_o + N_c)\, 2^{N_d}\right)$.

6.3 Incremental method

So far we have described (naive) methods based on the compilation of a full SDD out of all the relevant decision variables and probabilistic variables. In this chapter we describe a method that builds up SDDs out of probabilistic variables only, using a method that is inspired by $T_\mathcal{P}$-compilation [39] and based on depth-first search: INCREMENTAL.

Introductory observations on how to keep SDDs small

We observe that solving the problem involves trying out different strategies, i.e. different assignments to decision variables. The strategies described in Chapter 6.1 simply change the weight of the decision variables in the compiled SDD to evaluate the result of different strategies. The results in Figure 5 show that compiling SDDs becomes infeasible when problems consist of about 150 variables or more. We see an opportunity of reducing overall solving times (preprocessing plus search) by building SDDs out of probabilistic variables only, thus keeping them smaller and therefore quicker to compile.

Building SDDs solely out of probabilistic variables comes at the price of having to build an SDD for each strategy under consideration. Given a certain strategy $\sigma = \{(d_1 \mapsto b_1), \ldots, (d_{N_\delta} \mapsto b_{N_d})\}$, with N_d decision variables d_i and $b_i \in \mathbb{B}$, the decision variables d_i in the DAG representation of a query can be substituted by constants representing their truth values ($\top$ and $\bot$), through a substitution $\theta = \{d_i/b_i \mid (d_i \mapsto b_i) \in \sigma\}$ (see also Appendix A.2). The newly obtained DAGθ can now be used to build an SDD in the same bottom-up fashion as described in Chapter 3.2. The resulting SDD contains only probabilistic variables (and, of course, constants $\top$ and $\bot$). Therefore, it is generally smaller than an SDD compiled out of both probabilistic and decision variables, simply because it contains fewer variables.

Generally, the resulting SDD is also smaller for another reason: consider an AND-node in the DAG that has at least one decision variable d amongst its child nodes, representing the following formula: $f_{AND} = c_1 \wedge \cdots \wedge d_i \wedge \cdots \wedge c_n$. In the compiled SDD, there is a sub-SDD that represents the formula that is represented by this AND-node, and this sub-SDD has a non-zero size (because at the very least, it needs to represent the two possible choices for the value of d). Now suppose d is assigned value $\bot$ in strategy σ. This means that $f_{AND} \models \bot$: the AND-node in the DAG represents a formula that evaluates to $\bot$, no matter what values the other children of the AND-node in the DAG have. The sub-SDD can then be replaced by the terminal node $\bot$, which has size zero. A similar argument can be made for OR-nodes in the DAG.

Consider as an example the `mes` problem set. An edge (u, v) in this set is represented by a decision variable d_{uv} and a probabilistic variable e_{uv}. In order for the edge to exist, both must be true, and an SDD that represents the existence of this edge has size 2. By representing this edge by its probabilistic edge only, we compile a zero-sized SDD consisting of the terminal representing e_{uv} in case $d_{uv} \mapsto \top$ in strategy σ, or a zero-sized SDD consisting of the terminal that represents $\bot$ in case $(d_{uv} \mapsto \bot)$ in σ. Either way, the result is a smaller SDD than if both decision variables and probabilistic variables were used to build the SDD.

The incremental part of this strategy comes from the fact that search is based on the depth-first search method described in Chapter 6.1. The search starts with all decision variables set to $\bot$, causing the entire SDD to be deterministically $\bot$, and thus having size 0, for each query. During the search, decision variables are set to $\top$ one by one, causing some of the sub proofs in the program to no longer be deterministically $\bot$: they can be represented by small SDDs that live in the *SDD manager* (see also Appendix B.3). A particular SDD representing a query `path(s,t)` only stops being deterministically $\bot$ when a path from source `s` to target `t` is found.

(Example) execution of INCREMENTAL algorithm

The pseudocode of the INCREMENTAL method is given in Algorithm 4. Contrary to the DFS and EXHAUSTIVESEARCH methods, the INCREMENTAL algorithm requires as input only the compiled AND/OR DAG and the roots of the sub DAGs that represent queries: SDDs for the queries are built incrementally during search. Figure 10 shows an example of a compiled DAG that is used as input for the INCREMENTAL method. Figure 11 visualises what happens during the execution of the INCREMENTAL algorithm in two nodes of the search tree. The example problem in Figures 10 and 11 is an instance of the `mes` problem set of Chapter 5.

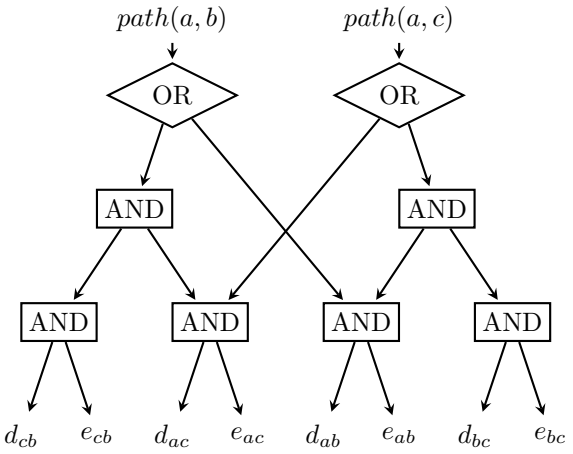


Figure 10: The AND/OR DAG for the queries $path(a,b)$ and $path(a,c)$ in the graph shown in Figure 6. The DAG has two roots that represent formulas $f_{ab} = (d_{ab} \wedge e_{ab}) \vee ((d_{cb} \wedge e_{cb}) \wedge (d_{ac} \wedge e_{ac}))$ and $f_{ac} = (d_{ac} \wedge e_{ac}) \vee ((d_{ab} \wedge e_{ab}) \wedge (d_{bc} \wedge e_{bc}))$. DAG and roots are used as input for the INCREMENTAL method.

In the initialisation phase of the INCREMENTAL algorithm, an *SDD manager* $\mathbf{M}$ is initialised with all the terminal SDD nodes that may be necessary to build larger SDDs: one terminal node for each probabilistic variable, plus nodes for $\top$ and $\bot$ (line 4). During the execution of the algorithm, we build small SDDs whenever necessary, each of which is associated with a node in the AND/OR DAGθ (shown in Figure 11 below the search tree). Here, DAGθ_j contains formulas $f\theta_j$, substituting decision variables for their truth values in the choice (C, a partial assignment of truth values to decision variables) that is currently under consideration in search tree node j. INCREMENTAL uses an array $\mathbf{A}$ of pointers to link each node in the DAG to the root of a sub SDD in the SDD manager that represents the same formula as the sub DAG rooted at the DAG node. Note that this sub SDD might only be a single terminal node, for example $\bot$ or a probabilistic variable. As with the DFS method, the INCREMENTAL method starts with all decision variables set to $\bot$, so this is how the array is initialised (line 7 in Algorithm 4).

In the root of the search tree, INCREMENTAL branches on $\top$ for the first variable: d_{ab} in Figure 11, branching to node i. The figure shows what happens in this visit tot node i: because now $d_{ab} \mapsto \top$, a sub proof in the DAG can be reconstructed with an SDD: $d_{ab} \wedge e_{ab} = \top \wedge e_{ab} = e_{ab}$, so node 11 in the DAG now points to the SDD consisting of only the terminal node e_{ab}. One of its parents, OR-node 15 in the left DAG in Figure 11, now also stops being deterministically $\bot$, and also points to e_{ab}. This updating of the DAG-node-to-SDD-root happens in line 28 of Algorithm 4. Observe that the SDD manager still only contains terminal nodes, each of which has size 0, so the total size of all SDDs in the SDD manager is still 0.

In the next node visit (to node *ii* in Figure 11), d_{ac} is set to $\top$, causing another AND-node in the DAG (node 10) and its parent OR-node 16 to point to e_{ac}. This is shown in the right DAG of Figure 11 and its associated DAG-node-to-SDD-root array and SDD manager. Again, the total size of all SDDs in the SDD manager is 0.

Suppose that $P(path(a,b)) \rightarrow \max$ is the optimisation criterion that maximises the probability for a path from a to b, and the constraint is that the probability for a path from a to c should not exceed 0.33 ($P(path(a,c)) \leq 0.33$). Checking the constraint now yields the conclusion that the partial assignment (choice) $C_{ii} = \{(d_{ab} \mapsto \top), (d_{ac} \mapsto \top)\}$ cannot lead to a strategy that respects the constraints, because

$P(e_{ac}) = 0.5 > 0.33$. This means that the search tree can be pruned at this point without having build any SDDs yet, for either optimisation criterion or constraint query.

Had the constraint been less strict, e.g. $P(path(a, c)) \leq 0.9$, the search would have had to continue by branching on $d_{bc} \mapsto \top$. This would cause the first non-zero-sized SDD to be built in the SDD manager (line 28), representing the formula $(d_{ab} \wedge e_{ab}) \wedge (d_{bc} \wedge e_{bc}) = e_{ab} \wedge e_{cb}$, which corresponds to node 14 in the DAG of Figure 11. Node 14's parent, node 16, can also be updated, and an SDD is built out of sub SDDs in the SDD manager so node 16 can be linked to (the root of) an SDD representing the formula $e_{ac} \vee (e_{ab} \wedge e_{cb})$.

Note that in our implementation, SDDs may be compiled several times, each in different branches of the search tree, because we save the current state of the SDD manager in each node before updating it and branching on 'true', and use that saved manager to later branch on 'false'.

Notes on complexity of the INCREMENTAL method

In terms of the number of search tree node visits, the INCREMENTAL method has the same complexity as DFS. The methods differ in initialisation complexity: the DFS method compiles a big SDD before the search begins, and the INCREMENTAL method compiles no SDDs during initialisation, but small ones during search. In each search tree node n_i (at depth i) whose corresponding choice (partial strategy) C_i satisfies the constraints, the SDD manager $\mathbf{M}$ is updated for $C_{i+1} \leftarrow C_i \cup \{d_{i+1} \mapsto \top\}$, before branching on 'true' for decision variable d_{i+1}. This requires a 'sweep' through part of the DAG (the part that corresponds to the nodes that are ancestors of d_{i+1}), to update the SDDs such that they reflect the (partial) formulas in DAGθ_{i+1}. Therefore complexity increases with the size of the DAG, as well the size of the SDDs that are compiled during search.

Algorithm 4 Depth-first search with incrementally compiled SDDs. The CheckConstraints method is that of Algorithm 2, the WMC method is analogous to Algorithm 1. Abusing notation, we use the same symbol both for the sub diagram in a DAG rooted at that node, and for the node itself. Similarly, we use the same symbol both for the root of an SDD and for the sub diagram rooted at that root.

Input: a compiled DAG with a set of roots of sub DAGs that encode logic formulas representing the probabilistic constraints, each with a corresponding threshold: $(f_c, \vartheta_c) \in \mathscr{C}$. Also a root of the sub DAG representing an optimisation criterion f_o.
Output: solution to constrained optimisation problem

1: **procedure** Incremental(DAG, $\mathscr{C}$, f_o)
2: DAG, $\mathscr{C}$, f_o ▷ Global constants
3: $\mathscr{F} \leftarrow \{f_c \mid (f_c, \vartheta_c) \in \mathscr{C}\} \cup \{f_o\}$ ▷ Global constant
Initialise SDD manager:
4: $\mathbf{M} \leftarrow \{p \mid p \in \mathscr{P}_{\mathscr{F}}\} \cup \{\top, \bot\}$ ▷ p is a probabilistic variable
Initialise DAG-node-to-SDD-root array, $\mathbf{A}[n]$ represents (the root of) the (sub) SDD in $\mathbf{M}$ that represents the same formula as the (sub) DAG rooted at n:
5: **for** node $n \in DAG$ **do**
6: **if** $n \in \mathscr{P}$ **then** $\mathbf{A}[n] \leftarrow n$
7: **else** $\mathbf{A}[n] \leftarrow \bot$
8: **end if**
9: **end for**
10: **return** Branch($\varnothing$, 0, $\mathbf{M}$, $\mathbf{A}$, 0, $\varnothing$)
11: **end procedure**

12: **procedure** Branch(C_δ, δ, $\mathbf{M}$, $\mathbf{A}$, $v_{\max}$, σ_{sol})
Base case: all decision variables have been assigned a value
13: **if** $\delta = N_d$ **then**
14: $\sigma \leftarrow C_\delta \cup \{(d \mapsto \bot) \mid d \in \mathscr{D}_{\mathscr{F}}, (d \mapsto b) \notin C_\delta\}$
15: $\theta \leftarrow \{d/b \mid (d \mapsto b) \in \sigma\}$
16: $\mathbf{M}, \mathbf{A} \leftarrow$ UpdateManager($\mathbf{M}$, $\mathbf{A}$, C_δ, $DAG\theta$)
17: $v_o \leftarrow \text{WMC}(\mathbf{A}[f_o], \sigma)$
18: **if** $v_o > v_{\max}$ **then**
19: $c_{\text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f], \vartheta) \mid (f, \vartheta) \in \mathscr{C})\}$, σ)
20: **if** c_{ok} **then** $v_{\max} \leftarrow v_o$, $\sigma_{\text{sol}} \leftarrow \sigma$ **end if**
21: **return** $v_{\max}$, σ_{sol}
22: **end if**

Recursive case: some decision variables remain unassigned
23: **else**
24: $c_{\text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f], \vartheta) \mid (f, \vartheta) \in \mathscr{C})\}$, σ))
25: **if** c_{ok} **then**
Branch on $\top$
26: $C_{\delta+1} \leftarrow C_\delta \cup \{(d_{\delta+1} \mapsto \top)\}$
27: $\theta \leftarrow \{d/b \mid (d \mapsto b) \in C_{\delta+1}\} \cup \{d/\bot \mid (d \mapsto b) \notin C_{\delta+1}, d \in \mathscr{D}_{\mathscr{F}}\}$
28: $\mathbf{M}_{\text{update}}, \mathbf{A}_{\text{update}} \leftarrow$ UpdateManager($\mathbf{M}$, $\mathbf{A}$, $C_{\delta+1}$, $DAG\theta$)
29: $v_{\max}$, $\sigma_{\text{sol}} \leftarrow$ Branch($C_{\delta+1}$, $\delta+1$, $\mathbf{M}_{\text{update}}$, $\mathbf{A}_{\text{update}}$, $v_{\max}$, σ_{sol})

Branch on $\bot$
30: $C_{\delta+1} \leftarrow C_\delta \cup \{(d_{\delta+1} \mapsto \bot)\}$
31: $v_{\max}$, $\sigma_{\text{sol}} \leftarrow$ Branch($C_{\delta+1}$, $\delta+1$, $\mathbf{M}$, $\mathbf{A}$, $v_{\max}$, σ_{sol})
32: **end if**
33: **end if**
34: **return** $v_{\max}$, σ_{sol}
35: **end procedure**

Algorithm 4 Continuation of INCREMENTAL pseudocode.

procedure UPDATEMANAGER($\mathbf{M}$, $\mathbf{A}$, C, $DAG\theta$)
  $Q \leftarrow \mathscr{D}_{\mathscr{F}}$ ▷ queue of nodes in $DAG\theta$ whose associated SDD may have to be updated
  while $Q \neq \varnothing$ **do**
    $n \leftarrow \text{pop}(Q)$
    if $(n \mapsto b) \in C$ **then**
      $\mathbf{A}[n] \leftarrow b$ ▷ $b \in \mathbb{B}$
    else if n is disjunction **then**
      $n \leftarrow \text{OR}(\{\mathbf{M}[n_c] \mid n_c \in \text{DAGCHILDREN}(n, DAG\theta)\})$
    else if n is conjunction **then**
      $n \leftarrow \text{AND}(\{\mathbf{M}[n_c] \mid n_c \in \text{DAGCHILDREN}(n, DAG\theta)\})$
    end if
    $Q \leftarrow Q \cup \{n_p \mid n_p \in \text{DAGPARENTS}(n, DAG\theta), n_p \notin Q\}$
  end while
  return $\mathbf{M}, \mathbf{A}$
end procedure

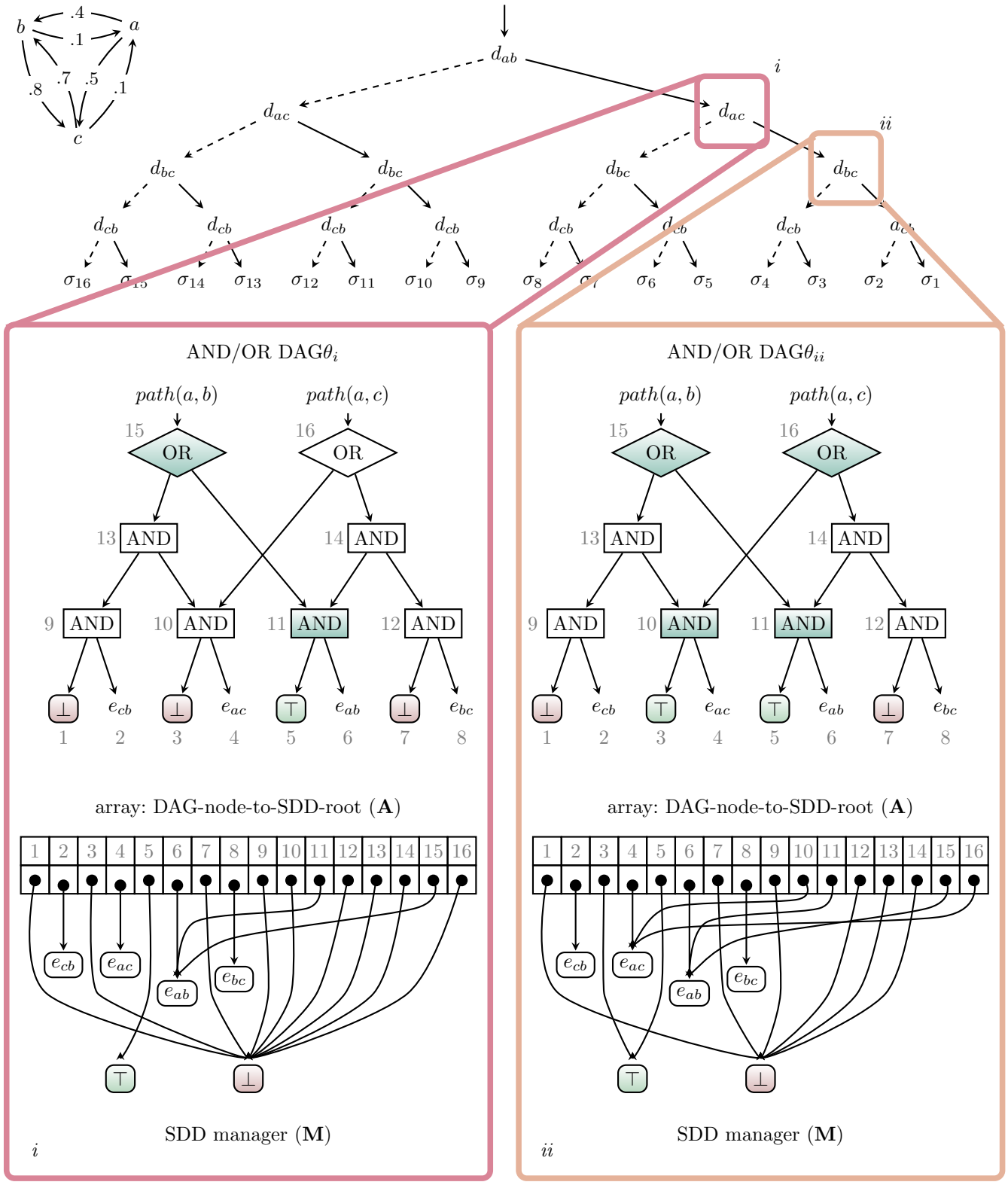


Figure 11: Example of two node visits in the execution of the INCREMENTAL method described in Chapter 6.3 for the example problem of the `mes` set of example problems in Figure 6. The graph representing the problem is repeated in the top part of this figure. The two queries under consideration are $path(a, b)$ and $path(a, c)$ (where we only consider simple paths). In the upper part of the figure we show the search tree over all decision variables of the problem. For two node visits in the search tree (labelled i and ii) the state of the DAG and and SDD manager are shown, with the array mapping each node of the corresponding DAGs to the root of an SDD in the SDD manager. The DAG nodes are labelled with their indices in the array. A decision variable for an edge (u, v) is denoted by d_{uv}, while e_{uv} denotes the corresponding probabilistic variable. The coloured internal nodes in the DAG indicate those that are not deterministically $\bot$.

6.4 Lazy incremental method

The INCREMENTAL method in the previous chapter is designed to postpone the compilation of (complex) SDDs as long as possible, and only compile them when it is absolutely necessary, thus saving unnecessary compilation time. The next method we propose takes this concept a step further. Recall the notion that only bounds for the probabilities of constraint queries are needed, no exact values. For the INCREMENTAL method we achieve that by assuming all unassigned decision variables to be $\bot$ during the search, computing lower bounds for the probabilities of the constraint queries.

We propose a LAZYINCREMENTAL method, based on the INCREMENTAL method, in which we (temporarily) assume all *probabilistic variables* to be *deterministically true*. The pseudocode is given in Algorithm 5. In the initialisation phase, all probabilistic variables are substituted by $\top$ (lines 7 and 11 in Algorithm 5). Because of this assumption of determinism, SDDs are kept very small during the search. After all: near the root of the search tree many variables will either be deterministically $\bot$ (decision variables) or deterministically $\top$ (probabilistic variables), causing most conjunctions and disjunction to also be deterministically $\bot$ or $\top$. This yields an *overestimation* of the *lower bound* of the probabilities of the queries. When this estimate exceeds the threshold ϑ for one of the constraints, the temporarily deterministic probabilistic variables need to be reset to their true probabilistic variables (lines 26 and 36, REMOVEDETERMINISMFROMCONSTRAINTS). This will either yield SDDs representing queries whose probabilities do not exceed ϑ, in which case the search can continue, or the lower bound will turn out to be too high, and the search can be pruned.

Removing (some of the) temporary determinism from the SDDs that represent the constraint queries happens as follows. The algorithm loops over the constraints that are violated and resets temporarily deterministic probabilistic variables that are relevant to that constraint one by one to their true probabilistic variables, until either the constraint is no longer violated, or the constraint contains no more temporarily deterministic probabilistic variables. The LAZYINCREMENTAL method always selects the temporarily deterministic probabilistic variable with the smallest probability to reset to its true value first, hoping to need few of these resets to find that the constraint can still be satisfied. Resetting a temporarily deterministic variable to a probabilistic variable cannot increase the estimate for the value of the probability of the constraint query. When the estimate for the probability of a constraint query drops below the corresponding threshold, the temporarily deterministic probabilistic variables that are relevant solely to that constraint query, are not reset during this round of removing temporary determinism. Note that it is not always necessary to remove all of the temporary determinism from an SDD that represents an constraint: if the overestimation of the lower bound for the probability of the constraint query drops below the threshold, we continue search. We can only be certain that a partial strategy violates a constraint if all the temporary determinism is removed from the SDD that represents that constraint. Therefore, in case of violated constraints, all temporary determinism does have to be removed from the SDD.

Observe that in lines 53 and 68 the SDD manager is updated in a way that is similar to the one described in UPDATEMANAGER in Algorithm 4: if a temporarily deterministic probabilistic variable is reset to its true probabilistic value, this change is propagated through the entire SDD manager, since it is updated in a bottom-up fashion in UPDATEMANAGER (in Algorithm 4). This means that coincidentally, by removing temporary determinism from the SDD manager because of the violation of one constraint, other violated constraints may become non-violated.

Note that while it may be sufficient to remove *some* of the temporary determinism from the constraints to know if pruning is possible or not, *all* temporary determinism needs to be removed from the SDD that represents the optimisation query in order to know if a new optimum is found. Since generally not all probabilistic variables are needed for the proof of the the optimisation query, some probabilistic variables may remain temporarily deterministic in this process, because they are relevant only to one or more constraints, even though all the determinism is removed from the SDD representing the optimisation query. The resetting of temporarily deterministic variables that are relevant solely to the optimisation criterion only happens in leafs of the search tree. Here an exact value for the probability of the optimisation query is needed (REMOVEDETERMINISMFROMOPTIMISATION in Algorithm 5), if the overestimation of the lower bound in that leaf is larger than the best (true!) value found so far. This is done in a way that is similar to that of removing temporary determinism from constraints: by simply looping over all relevant temporarily deterministic probabilistic variables and resetting them in the SDD manager to their probabilistic values. When the estimate drops below the best found value, LAZYINCREMENTAL can stop removing the temporary determinism in the probabilistic variables relevant to the optimisation criterion, and backtrack. Hence, it is likely that LAZYINCREMENTAL only removes all determinism from the SDD

Algorithm 5 Depth-first search with incrementally compiled SDDs, lazily making probabilistic variables temporarily deterministically $\top$. The CheckConstraints method is that of Algorithm 2, the WMC method is analogous to Algorithm 1, UpdateManager is defined in Algorithm 4. Abusing notation, we use the same symbol both for the sub diagram in a DAG rooted at that node, and for the node itself. Similarly, we use the same symbol both for the root of an SDD and for the sub diagram rooted at that root.

Input: a compiled DAG with a set of roots of sub DAGs that encode logic formulas representing the probabilistic constraints, each with a corresponding threshold: $(f_c, \vartheta_c) \in \mathscr{C}$. Also a root of the sub DAG representing an optimisation criterion f_o.
Output: solution to constrained optimisation problem

procedure LazyIncremental(DAG, $\mathscr{C}$, f_o)
  DAG, $\mathscr{C}$, f_o ▷ Global constants
  $\mathscr{F} \leftarrow \{f_c \mid (f_c, \vartheta_c) \in \mathscr{C}\} \cup \{f_o\}$ ▷ Global constant
  Initialise SDD manager:
  $\mathbf{M} \leftarrow \{p \mid p \in \mathscr{P}_{\mathscr{F}}\} \cup \{\top, \bot\}$ ▷ p is a probabilistic variable
  Initialise DAG-node-to-SDD-root array, $\mathbf{A}[n]$ *represents (the root of) the (sub) SDD in* $\mathbf{M}$ *that represents the same formula as the (sub) DAG rooted at* n*:*
  for node $n \in DAG$ **do**
    if $n \in \mathscr{P}$ **then** $\mathbf{A}[n] \leftarrow n$
    else if $n \in \mathscr{D}$ **then** $\mathbf{A}[n] \leftarrow \top$
    else $\mathbf{A}[n] \leftarrow \bot$
    end if
  end for
  Initially, all probabilistic variables are temporarily set to a deterministic $\top$*:*
  $\theta_p \leftarrow \{p/\top \mid p \in \mathscr{P}_{\mathscr{F}}\}$
  return Branch($\varnothing$, 0, $\mathbf{M}$, $\mathbf{A}$, 0, $\varnothing$, θ_p)
end procedure

procedure Branch(C_δ, δ, $\mathbf{M}$, $\mathbf{A}$, $v_{\max}$, σ_{sol}, θ_p)
  Base case: all decision variables have been assigned a value
  if $\delta = N_d$ **then**
    $\sigma \leftarrow C_\delta \cup \{(d \mapsto \bot) \mid d \in \mathscr{D}_{\mathscr{F}}, (d \mapsto b) \notin C_\delta\}$
    $\theta_d \leftarrow \{d/b \mid (d \mapsto b) \in \sigma\}$
    $\mathbf{M}$, $\mathbf{A} \leftarrow$ UpdateManager($\mathbf{M}$, $\mathbf{A}$, C_δ, $DAG\theta_d$)
    $v_o \leftarrow \text{WMC}(\mathbf{A}[f_o], \sigma)$
    while $v_o > v_{\max}$ and SDD rooted at $\mathbf{A}[f_o]$ contains temporary determinism **do**
      $\mathbf{M}$, $\mathbf{A}$, `newopt`, v_o, $\theta_p \leftarrow$ RemoveDeterminismFromOptimisation($\mathbf{M}$, $\mathbf{A}$, v_o, $v_{\max}$, DAG, θ_d, θ_p, σ)
    end while
    if `newopt` **then**
      $c_{\text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f], \vartheta) \mid (f, \vartheta) \in \mathscr{C}\}$, σ)
      if not c_{ok} **then**
        $\mathbf{M}$, $\mathbf{A}$, θ_p, $c_{\text{ok}} \leftarrow$ RemoveDeterminismFromConstraints($\mathbf{M}$, $\mathbf{A}$, $\mathscr{C}$, DAG, θ_d, θ_p)
      end if
      if c_{ok} **then** $v_{\max} \leftarrow v_o$, $\sigma_{\text{sol}} \leftarrow \sigma$ **end if**
    end if
    return $v_{\max}$, σ_{sol}

Algorithm 5 Continuation of LazyIncremental pseudocode.

Recursive case: some decision variables remain unassigned
else
  $\sigma \leftarrow C_\delta \cup \{(d \mapsto \bot) \mid d \in \mathscr{D}_\mathscr{F}, (d \mapsto b) \notin C_\delta\}$
  $\theta_d \leftarrow \{d/b \mid (d \mapsto b) \in \sigma\}$
  $c_{\text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f], \vartheta) \mid (f, \vartheta) \in \mathscr{C}\}$, σ)
  if not c_{ok} **then**
    $\mathbf{M}$, $\mathbf{A}$, θ_p, $c_{\text{ok}} \leftarrow$ RemoveDeterminismFromConstraints($\mathbf{M}$, $\mathbf{A}$, $\mathscr{C}$, DAG, θ_d, θ_p)
  end if
  if c_{ok} **then**
    Branch on $\top$
    $C_{\delta+1} \leftarrow C_\delta \cup \{(d_{\delta+1} \mapsto \top)\}$
    $\mathbf{M}_{\text{update}}$, $\mathbf{A}_{\text{update}} \leftarrow$ UpdateManager($\mathbf{M}$, $\mathbf{A}$, $C_{\delta+1}$, $DAG\theta_d$)
    $v_{\max}$, $\sigma_{\text{sol}} \leftarrow$ Branch($C_{\delta+1}$, $\delta+1$, $\mathbf{M}_{\text{update}}$, $\mathbf{A}_{\text{update}}$, $v_{\max}$, σ_{sol}, θ_p)

    Branch on $\bot$
    $C_{\delta+1} \leftarrow C_\delta \cup \{(d_{\delta+1} \mapsto \bot)\}$
    $v_{\max}$, $\sigma_{\text{sol}} \leftarrow$ Branch($C_{\delta+1}$, $\delta+1$, $\mathbf{M}$, $\mathbf{A}$, $v_{\max}$, σ_{sol}, θ_p)
  end if
end if
return $v_{\max}$, σ_{sol}
end procedure

procedure RemoveDeterminismFromOptimisation($\mathbf{M}$, $\mathbf{A}$, f_o, v, $v_{\max}$, DAG, θ_d, θ_p, σ)
  while f_o contains temporary determinism and $v > v_{\max}$ **do**
    $p \leftarrow$ SelectVariable(f_o) ▷ $p \in \mathscr{P}_\mathscr{F}$ is temporarily deterministic and relevant to f_o
    $\theta_p \leftarrow \theta_p \setminus \{(p/\top)\}$
    $\theta \leftarrow \theta_d \cup \theta_p$
    $\mathbf{M}$, $\mathbf{A} \leftarrow$ RemoveDeterminismFromManager($\mathbf{M}$, $\mathbf{A}$, $DAG\theta$)
    $v \leftarrow$ WMC(f_o, σ)
  end while
  if $v > v_{\max}$ and f_o contains no determinism **then** **return** $\mathbf{M}$, $\mathbf{A}$, `True`, v, θ_p
  else if $v \leq v_{\max}$ and f_o contains no determinism **then** **return** $\mathbf{M}$, $\mathbf{A}$, `False`, v, θ_p
  else if $v \leq v_{\max}$ and f_o contains determinism **then** **return** $\mathbf{M}$, $\mathbf{A}$, `False`, v, θ_p
  end if
end procedure

procedure RemoveDeterminismFromConstraints($\mathbf{M}$, $\mathbf{A}$, $\mathscr{C}$, DAG, θ_d, θ_p, σ)
  for $(f_c, \vartheta) \in \mathscr{C}$ **do**
    $c_{f_c\ \text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f_c], \vartheta)\}$), σ
    while f_c contains temporary determinism and not $c_{f_c\ \text{ok}}$ **do**
      $p \leftarrow$ SelectVariable(f_c) ▷ $p \in \mathscr{P}_\mathscr{F}$ is temporarily deterministic and relevant to f_c
      $\theta_p \leftarrow \theta_p \setminus \{(p/\top)\}$
      $\theta \leftarrow \theta_d \cup \theta_p$
      $\mathbf{M}$, $\mathbf{A} \leftarrow$ RemoveDeterminismFromManager($\mathbf{M}$, $\mathbf{A}$, $DAG\theta$)
      $c_{f_c\ \text{ok}} \leftarrow$ CheckConstraints($\{(\mathbf{A}[f_c], \vartheta)\}$, σ)
    end while
    if not $c_{f_c\ \text{ok}}$ **then return** $\mathbf{M}$, $\mathbf{A}$, `False` **end if**
  end for
  return $\mathbf{M}$, $\mathbf{A}$, `True`
end procedure

for the optimisation criterion if it finds a solution that is better than the best solution we found earlier. This means that only in these cases, a (relatively) big SDD for this query needs to be built, while the INCREMENTAL method requires the building of (relatively) large SDDs in each leaf that is reached during search.

Notes on complexity of LAZYINCREMENTAL method

The size of the search tree for the LAZYINCREMENTAL method is exactly the same as that of INCREMENTAL. The algorithms differ in the number of sweeps over (part of) the DAG that are needed to update the SDD manager. After all: these updates happen for the INCREMENTAL method at most once per visit to a search tree node, while LAZYINCREMENTAL also needs them each time one temporarily deterministic probabilistic variable is reviewed and made probabilistic again. Note that generally, these sweeps are less expensive for LAZYINCREMENTAL, since the SDDs that are created during the process are kept smaller, because of the temporary determinism of other probabilistic variables.

6.5 Optimisations for incremental methods

Although not explicitly stated in the pseudocodes of the INCREMENTAL and LAZYINCREMENTAL methods presented above, some optimisations can be applied to these algorithms. One is that of being more smart about updating the SDD manager. In our implementation of INCREMENTAL and LAZYINCREMENTAL, the queue Q with DAG nodes that may have to be updated is not initialised with *all* decision variables or (temporarily deterministic) probabilistic variables, but only with those whose substitution is different from their substitution the last time the SDD manager was updated. This way, only the part of the SDD manager that might need an update, is updated. This optimisation can be expanded by only compiling a new SDD for node n in DAGθ if the SDDs that represent n's children in the DAG have changed during this call to UPDATEMANAGER.

Further possible optimisations are discussed in Chapters 6.6 and 10.

6.6 Different optimisation and constraint settings

All the search methods described in this chapter can easily be adapted for the **maxSet** optimisation setting. We no longer need to compute the WMC for the optimisation criterion, but can simply count the number of decision variables that are set to $\top$. For the **maxSet** setting, we can perform an additional way of pruning: whenever the sum of decision variables that map to $\top$ in partial assignment C and the number of unassigned decision variables does not exceed the size of $\sigma_\top$ for the best solution found so far, the search can be pruned.

Note also that this optimisation setting does not require an SDD to be built for the optimisation criterion, and that some probabilistic variables may not be relevant to the constraints, and can thus remain deterministic in the LAZYINCREMENTAL method, limiting the size of the resulting SDDs even further.

The INCREMENTAL and LAZYINCREMENTAL methods in their current form cannot be used for optimisation criteria of the types **minProb** and **minSet**, combined with constraints of the form $P(q) \geq \vartheta$ yet. The reason was already touched upon in Chapters 6.1 and 6.3: it would require the computation of *upper bounds* for the constraint queries. Compiled SDDs could be kept small by assuming non-assigned decision variables to be $\bot$ when computing lower bounds for query probabilities, causing probabilistic variables to be no longer relevant and thus not included in the SDD. In order to compute an upper bound, however, unassigned decision variables should get value $\top$, which causes many probabilistic variables to be relevant after all, which yields large SDDs.

7 Experimental setup

In this chapter we specify the methods and settings used for the experimental part of this work. Please refer to Chapter 5 for a description of the artificially generated example sets of SCOPs used in this work. All experiments are performed on a 4-thread machine, with four Intel(R) Xeon(R) CPU E3-1225 v3 @ 3.20 GHz CPUs and 32 Gb memory, using the Python ProbLog library [15, 16], which uses the SDD library from the Automated Reasoning Group of UCLA [8], and the implementations of our own algorithms in Python 2.7.

7.1 Generating the artificial example problems

We introduced the sets of examples of instances of stochastic constraint programming in Chapter 5. In this chapter we provide more details on how these example sets were generated.

General graph generation parameters

As mentioned in Chapter 5, the example problems are all based on social networks generated with *gengraph* [38]. The bash script `gengraph` can generate random graphs whose degree distribution obeys the power law:

$$P(X = k) = k^{-\alpha}, \tag{19}$$

with $P(X = k)$ the probability that a node X has degree k and α the parameter that represents the heavy-tail behaviour. The script ask us to specify α, the number of nodes N and a degree interval specifying what the minimum and maximum degrees are that should be found in the result. We choose $\alpha = 2.5$, choose the interval $[2, \ldots, N]$ as the degree interval and then try different numbers of nodes N. If the script is able to generate a graph fitting the α and degree interval requirements, we save it. For the `mes` and `fas` problem set, the number of edges determines the number of decision variables. Thus, we save graphs that have the desired number of edges. The examples for the `tyf` problem set are generated slightly differently. We generate a graph and add a node that will act as the source. The outgoing edges of this node are the decision variables. In our problem set, we add outgoing edges to half of the nodes in the graph. Thus, a graph generated by `gengraph` is saved if its number of nodes is twice the desired number of decision variables.

The `gengraph` script returns strongly connected, directed graphs with for each edge (u, v) also an edge (v, u). The source node of the `mes` problems is randomly chosen, weighted by the outdegree of the nodes in the network. For the `tyf` problem set the node with the largest outdegree is chosen as the source. The candidate targets are chosen such that there exists at least one path of length at least three and at most ten from source to target. The actual targets for the queries are selected randomly and uniformly.

The graphs that are generated by the `gengraph` script can be used almost directly for the `mes` problem set. All that is needed for the `mes` problems is to generate probabilities for the edges. The probabilities are drawn randomly and uniformly from $\{0.1, \ldots, 0.9\}$, where the probability associated with edge (u, v) may be different from the one associated with edge (v, u). For the `tyf` problems the extra source node and outgoing edges are added. Finally, probabilities for the other edges are generated and assigned in the same way as for the probabilistic edges in the `mes` problems. For the `fas` problem set, no alterations need to be made to the network, and no probabilities need to be generated, as the probabilities are fixed by the program specified in Chapter 5.

Characteristics and limitations of the artificial stochastic constraint optimisation instances

The graph generation tool `gengraph` is designed by Viger and Latapy as an algorithm for generating large graphs in $O(n \log n)$ time (with n the number of vertices). They present experiments done on graphs with $n = 10^4$ vertices; several orders of magnitude more than the graphs we generate. We can therefore not be certain of the quality and consistency of the test examples generated by this tool. We see this reflected in the diversity of graph topologies in our problem set.

As an example consider Figure 12. The figure shows three graphs from our `mes` problem set. The graphs show very different characteristics for, for example, closeness and betweenness centrality.

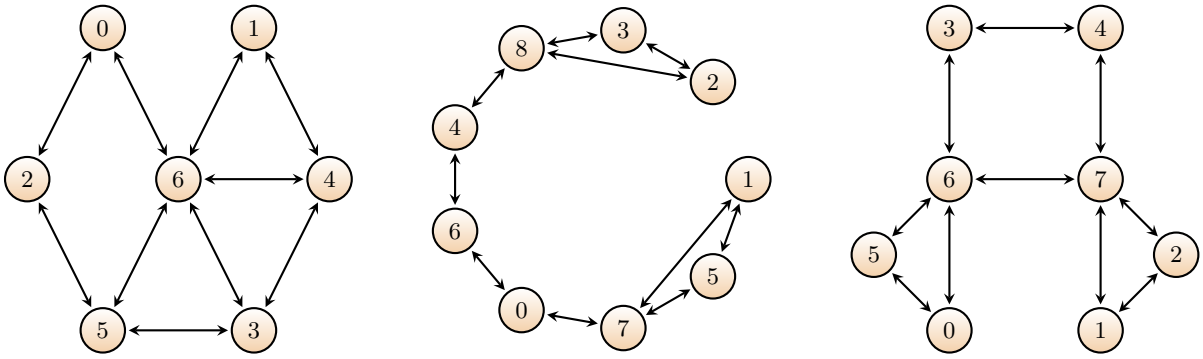


Figure 12: Some examples of graphs generated for by the `gengraph` tool for the `mes` problem set. These graphs correspond to problems with twenty decision variables and twenty probabilistic variables.

7.2 Establishing a benchmark for DAG and SDD compilation

In Chapter 4 we discussed the complexity of compiling logical formulas to DAGs and SDDs. We perform a simple experiment to establish at what size the SDDs become infeasible to compute, for our sets of example probabilistic constrained optimisation problems (the results of which can be found in Figure 5 in Chapter 4, instead of in Chapter 8).

We take a program with $N_v = N_d + N_p$ variables (N_d decision variables and N_p probabilistic variables) and $N_q = N_o + N_c$ optimisation and constraint queries. We compile the program into a DAG and measure its size. The size of a DAG is defined as its number of nodes (see Appendix B.1). We also measure the time needed to compile the DAG. Note that such the DAG has several roots: one for each query. These roots generally share descendants, so we expect the size of the DAG to depend both on the number of variables and on the number of queries.

Then we compile the logic formulas expressed by the DAG into SDDs. Similar to the DAG, we have an *SDD manager* that contains various sub SDDs (see Appendix B.3 and Chapter 6.3). Each query has its own root and may share sub-SDDs with other SDDs. We measure the total size of the SDD manager and that of the SDD for the first query. We expect the size of the SDD of the first query as well as the size of the SDD manager to depend both on the number of variables and on the number of queries.

Finally, we measure the time needed for the compilation of both the DAG and the SDD. We repeat the experiment for problems with different numbers of decision variables. Test problems always have a multiple of ten decision variables. For each number of decision variables, we repeat the experiment on ten different instances of such a problem. We do not aggregate over the data.

Specifically, we perform the DAG and SDD compilation benchmark on the `fas` problem set, with problems on numbers of decision variables ranging from twenty to sixty. The corresponding total numbers of variables range from 47 to 139 variables. We did not consider larger problems because compiling the SDD for the first problem on seventy decision variables was terminated after 48 hours, which we consider to be an indication that compiling the SDD for problems on seventy decision variables or more is unfeasable. We repeat the experiment for problems with one and five queries. Because this experiment only involves preprocessing, the optimisation type, constraint type and threshold ϑ are irrelevant to this experiment.

7.3 Measuring search times and other search characteristics

The focus of our work is on the search process rather than the preprocessing (compilation of DAG and, in case of the naive methods, compilation of the big SDD out both probabilistic and decision variables). Therefore most of our experiments are on search times. Again, we have ten problem instances for each number of decision variables. After initialising each search strategy, we measure the time needed to find a solution to the constrained optimisation problem. We use a timer on the experiments and stop each search after 3600 seconds (one hour), unless indicated otherwise.

We perform the experiments on the three artificially generated sets of SCOPs, for the **maxProb** optimisation setting, using one or three constraints of type $P(q) \leq \vartheta$, with q a query and different values of ϑ, the exact values of which depend on the problem type. The `mes` and `tyf` problems contain probabilistic variables ranging from 0.1 to 0.9, so we expect constraints to be likely to be violated (and respected) for a range of ϑ's. On the other hand, the `fas` problem contains the `0.3::stress(X) :-`

`person(X).` clause. This means that each query has a base success probability of 0.3, so any constraint of the form $P(\texttt{smokes(X)}) \leq \vartheta$ with $\vartheta < 0.3$ is automatically violated. We therefore need not look at ϑ's that are smaller that 0.3. Similarly, the probability of someone smoking is not that large. A simple lower bound estimate (assuming the probability of a friend smoking to be 0.3) gives that a person with one friend has a probability of being a smoker that is bounded below by 0.405. This increases to roughly 0.69 if they have five friends, and to roughly 0.86 if they have ten. The example problems being but small, people generally have not many friends, and we expect that for $\vartheta > 0.4$ it is easy to find solutions that satisfy the constraint. In experiments with more than one constraint, we use the same value for ϑ for each of the constraints.

The experiments that evaluate search times and search behaviour for ExhaustiveSearch, dfs, Incremental and LazyIncremental methods, we perform experiments on small example problems only. Specifically, **maxProb** experiments on the `mes` and `fas` problem sets are done on problems of twenty and thirty decision variables. The **maxProb** type experiments on the `tyf` example instances are done on problems with only ten decision variables, as these problems contain more probabilistic variables than the `mes` and `fas` problems, causing the query evaluations during search to be slower.

Because the search times for the ExhaustiveSearch search method do not vary much, as is expected based on the nature of the strategy, we benchmark this method once for a single threshold only (specifically $\vartheta = 0.3$), and compute the mean search time μ_s and corresponding standard deviation σ_s. In Chapter 8 we present μ_s and the 90% confidence interval ($\pm 0.96\sigma_s$). If one or more of the searches were timed out, only the mean search time of the remaining examples is given, along with the number of timed-out examples.

For all the other search methods (each based on depth-first search), we benchmark on different values for ϑ. We expect search times to be lower for small values of ϑ, because of pruning due to partial strategies that violate the constraints. We also expect search times to decrease for large values of ϑ, because of the application of sssp.

In order to gain more insight in the pruning, we record the number of nodes visited at each level of the search tree, for all DFS-based strategies. We compare this to the number of possible strategies, because the ExhaustiveSearch method evaluates that number of possible strategies (2^{N_d}, see Chapter 6.1). The strategies differ in when and how constraint queries and optimisation criteria are evaluated, and this also depends highly on the problem and, for example, the order in which decision variables are branched on. However, we can still use this number as a measure for the effectiveness of the pruning. Note that the dfs, Incremental and LazyIncremental methods each traverse the search tree in the same order (and visit the same nodes).

The Incremental and LazyIncremental methods each attempt to keep the compiled SDDs as small as possible. In order to gauge their performance, we measure the size of the largest SDD created during search for each query, as well as the average size of the SDDs compiled for that query. We expect the size of the largest SDD to be a bottleneck for the performance of the algorithm. The mean size of SDDs is an indication of how large the SDDs are during search. We expect the SDDs to only grow large near the leaves of the search tree. We compare these sizes to the size that the SDD would have, was it compiled out of both probabilistic and decision variables for the naive methods.

Furthermore, we record for each query what the largest size of the SDD representing that query is on each path from root to leaf in the search tree. We expect the distribution of largest SDD sizes to provide an indication of the possible efficiency of a greedy and non-exhaustive version of the Incremental and LazyIncremental methods. After all, not only the size of the largest SDD that needs to be compiled is a bottleneck for the total search time, but also the number of times that a large SDD needs to be compiled.

Note that aside from the actual search times, the number of nodes visited in the search tree as well as the size of the SDDs provide information about the effectiveness of the different approaches. The last two, combined with additional information from the value of ϑ and details about the specific problems, can be used to gain insight in the characteristics of both the solving methods and the problems that make the methods effective (or less effective) for solving these kinds of problems.

8 Experimental results

In this chapter we present and discuss some of our results in order to answer the second and third research questions: how effective are the methods presented in Chapter 6 for the type of problem described in Chapter 4, and why? Because of the many parameters in our experiments (four different algorithms, two optimisation settings, three types of problems, different problem sizes, different values for ϑ, different numbers of constraints), we limit the results in this chapter to a selection. Some other results can be found in Appendix C. In this chapter we do mention the main conclusions we can draw from those results, but they are discussed in some more detail in Appendix C.

8.1 Compilation times of DAGs and SDDs

Recall the results presented in Figure 5. They show how the sizes of the DAGs representing queries and the corresponding SDDs and SDD manager increase with the number of variables (both decision variables and probabilistic variables) present in those DAGs and SDDs (Figures 5a and 5b). From the results we conclude that the size of the SDDs increases more rapidly with the number of variables than the size of the DAGs, and that the SDDs also tend to be larger than those of the DAGs in an absolute sense.

Figures 5c and 5d show the compilation times of DAGs and SDDs, respectively, as a function of their sizes. Again, we see that the compilation times for SDDs increase more rapidly than those for DAGs.

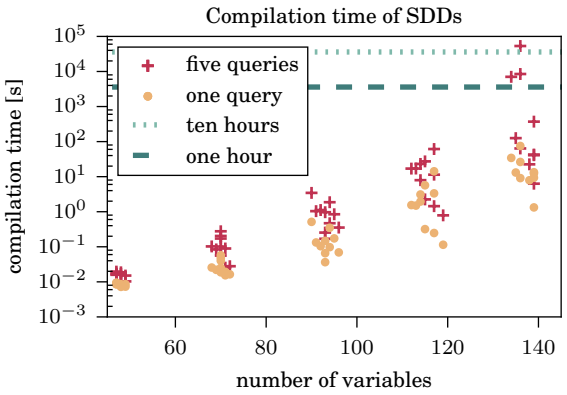


Figure 13: Compilation time of SDDs as a function of the number of variables (both probabilistic and decision). Data from examples in `fas` problem set.

Consider the results in Figure 13. We see that from roughly 140 variables onwards, there are examples for which the compilation time is in the order of hours. Add more variables and compiling an SDD becomes a matter of days. These results provide an indication of how much room there is to repeatedly build smaller SDDs. Recall from Chapter 4 and Chapter 6 that SDDs built by Incremental and LazyIncrementalare built out of roughly half the number of variables than the ones built by ExhaustiveSearch and dfs (or even less). From the results in Figure 13 we deduce that building an SDD from about seventy variables takes over five orders of magnitude less time than building one out of roughly 140 variables. A naive guess would led us to believe that this means that Incremental and LazyIncremental can build about 10^5 of these smaller SDDs before the total SDD building time exceeds that of the naive approach.

Of course, in practice it is not that simple. After all: SDDs are built incrementally, so the definition of the building time of one SDD becomes somewhat complicated. Furthermore, given the rather naive way in which these SDDs are built, SDDs building times also depend on the size of the corresponding DAGs and factors such as the order in which the search tree branches on decision variables, the order in which constraints are evaluated, and the order in which LazyIncremental sets temporarily deterministic variables to their true probabilistic values.

8.2 Comparison of search times

A first step towards evaluation the second research question, is to compare the search times of the different algorithms.

Search times for the maxProb optimisation setting

Let us first consider the **maxProb** setting. As a benchmark, search times for the ExhaustiveSearch and dfs methods are presented in Figures 14a and 14d for the `mes` and `tyf` problem sets. Each of these figures shows the search times needed by the dfs method for different problem instances against the value for ϑ. Additionally, the plots show a dotted line, which represents the time after which the search in one example was terminated. The dashed lines represent the mean search time μ_s that was needed by the

ExhaustiveSearch method for problems on twenty decision variables (Figure 14a) or ten decision variables (Figure 14d), for $\vartheta = 0.3$.

Given the mean search times for ExhaustiveSearch on problems of twenty (ten) decision variables, we can extrapolate these to mean search times for problems on thirty (twenty) decision variables. This yields mean search times of several hundred hours, so we have not performed those experiments. Note that the given 90% confidence interval for Figures 14d and 15d is not a typo. One of the optimisations implemented in ExhaustiveSearch (and the other algorithms) mentioned in Chapter 6.1 is that the search is stopped when the perfect score is found. Because of the way the graphs and queries are generated for `tyf` problems, it might happen that the target node in the optimisation query is a neighbour of the source node. In that case, choosing the (s, t) edge to be $\top$ yields a probability for $path(s, t)$ of 1: a perfect score. If this is part of a (partial) strategy that satisfies the constraints, the search is stopped. Therefore, search times of ExhaustiveSearch may vary a lot for problems of type `tyf` after all.

Looking at the results in Figure 14, we immediately see that the different problem sets show very different behaviour. As expected, we see that problems on fewer decision variables are generally solved in less time (within the same set of SCOP instances). We also observe a large variation in dfs's, Incremental's and LazyIncremental's search times for different examples with the same number of decision variables, the same threshold and the same number of constraints: there are problems that take several orders of magnitude less time to solve than others.

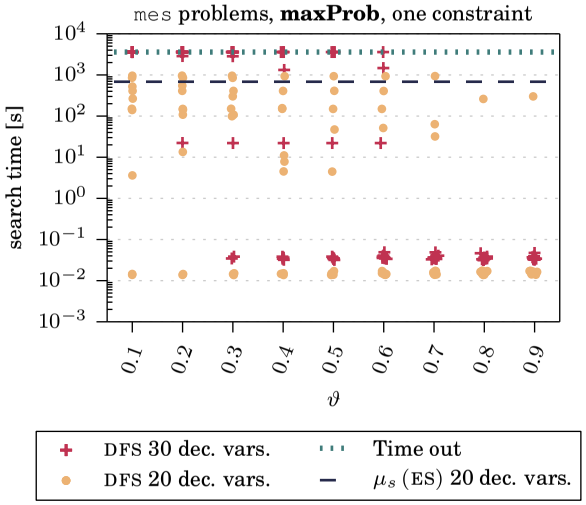
Looking at the behaviour for different types of example problems, we see some differences. We see that for the `mes` and `tyf` examples in Figure 14, search times may be high or low for different examples for all kinds of constraints. The results for `fas` in Figure 23 in Appendix C show that only for very specific values of ϑ, `fas` problems take a long time to solve. The results for the `mes` (and `fas`) problems do seem to confirm that sssp is an effective pruning method, as the problems with loose constraints (high values of ϑ) tend to be solved very quickly.

Now consider the search time results for the Incremental method in Figures 14b and 14e. The first observation is that the results look very similar to those presented for the dfs method, but there are some differences. For example: it seems that the Incremental method solves the `tyf` problems faster than the dfs method, especially for $\vartheta = 0.2$. That being said, the `mes` problems show slightly larger search times when solved by the Incremental method, than if they are solved by the dfs method. We also see that the variation in solving times seems to be a bit larger for the Incremental method than for the dfs method (this can be clearly seen when comparing the results in Figure 14a and Figure 14b). It is remarkable that search times for `tyf` problems tend to be lower for the Incremental method than for the dfs method, except that one example tends to time-out more often with the Incremental method than the dfs method (the problem instance that times out is in both cases the same one).

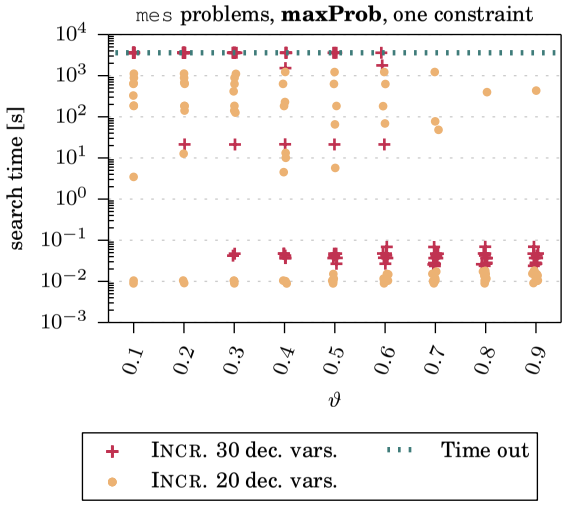
We can explain the result that the incremental methods tend to do better on the `tyf` examples than the dfs method by considering the nature of the `tyf` problems. Suppose one of the (partial) proofs in a SCOP is that for a path from a to b: $path(a, b)$. Recall from Chapter 5 that in the `mes` and `fas` problems, several decision variables need to be set to $\top$ in order to create a probabilistic path from one node to another: one decision variable for each edge on the path. Suppose one of the paths from a to b is the following: $path_{ab} = a \to u \to v \to w \to b$, with corresponding formula $f_{path_{ab}} = d_{au} \wedge e_{au} \wedge d_{uv} \wedge e_{uv} \wedge d_{vw} \wedge e_{vw} \wedge d_{wb} \wedge e_{wb}$ (with d a decision variable and e a probabilistic variable). As long as not all of the decision variables in $f_{path_{ab}}$ are chosen to be $\top$, $f_{path_{ab}} \models \bot$ holds. Therefore, while building the SDD that corresponds to that proof, the incremental algorithms have to traverse part of the DAG each time one of the decision variables on the path is chosen to be $\top$, to see if the expression for $path_{ab}$ changes, while $f_{path_{ab}} \models \bot$ unless $d_{au} = d_{uv} = d_{vw} = d_{wb} = \top$. Traversing part of the DAG to update formulas whose expression does not change, even though variables in the formula do change their values, takes time while nothing new is learned. Suppose that $path(a, b)$ is a constraint query, and now the algorithm finds out that with a non-$\bot$ proof for p_{ab}, the constraint is violated. That means that backtracking needs to take place and the algorithm has just wasted a lot of time.

On the other hand, the same query in a `tyf` examples would correspond to a formula $f_{path_{ab}} = d_{au} \wedge e_{uv} \wedge e_{vw} \wedge e_{wb}$. Therefore, with choosing $d_{au} = \top$, the algorithm immediately obtains a non-$\bot$ expression for p_{ab} and can immediately prune the search, if necessary.

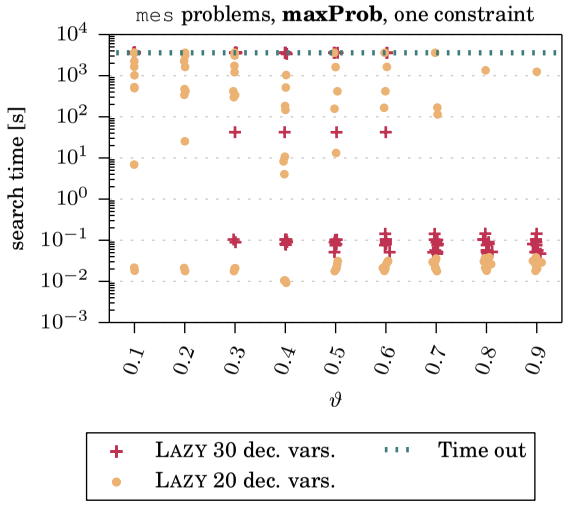
In other words: while for the `mes` (and `fas`) problem sets new information is typically only gained if sets of decision variables are set to $\top$, for `tyf` problems information is gained with only a single assignment of $\top$ to a decision variable. This results in less work for the incremental algorithms to be done on proofs for `tyf` problems, than for proofs on `mes` problems.



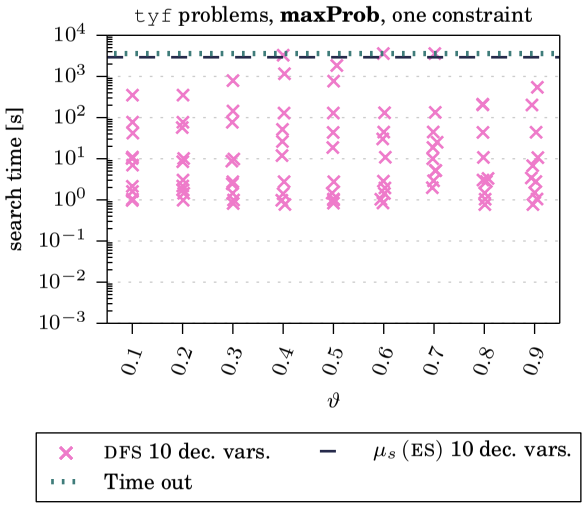
(a) ExhaustiveSearch and dfs, μ_s (es) $= 0.68 \pm 0.08 \cdot 10^3$ s.



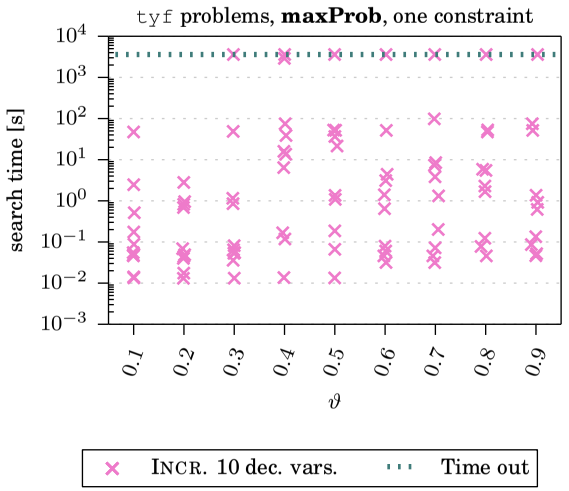
(b) Incremental, `mes` problem set, one constraint.



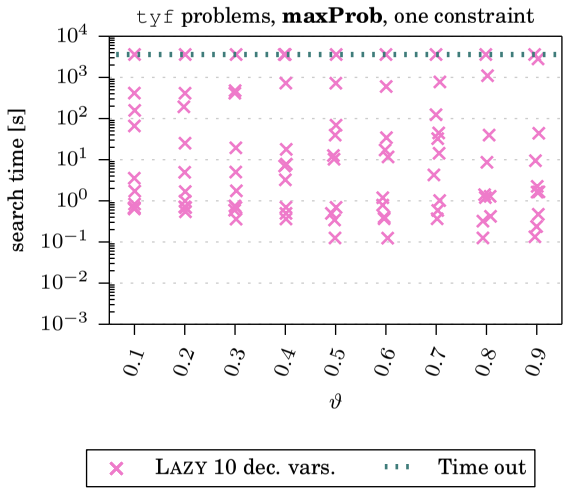
(c) LazyIncremental, `mes` problem set, one constraint.



(d) ExhaustiveSearch and dfs, μ_s (es) $= 3 \pm 13 \cdot 10^3$ s (one time-out).



(e) Incremental, `tyf` problem set, one constraint.



(f) LazyIncremental, `tyf` problem set, one constraint.

Figure 14: Search times for ExhaustiveSearch and dfs (left), Incremental (middle) and LazyIncremental (right) for `mes` problem set (top) and `tyf` problem set (bottom), in the **maxProb** setting with one constraint. Jitter added in horizontal direction to separate datapoints.

Search times for the maxSet optimisation setting

Let us now move on to the results for the **maxSet** optimisation setting. We expect that these problems are easier to solve. After all: they allow for extra pruning due to the fact that it is easy to determine whether or not it is still possible to improve on the current best solution, on the current path in the search tree.

The results in Figure 15 seem to confirm this. The figure shows results for problems with two constraints. On the one hand this means that the number of query evaluations done per possible strategy (for the ExhaustiveSearch algorithm) or search tree node (for the other algorithms) is comparable to that for the **maxProb** setting. On the other hand it also means that it is more likely that a constraint is violated by a (partial) strategy, which influences pruning.

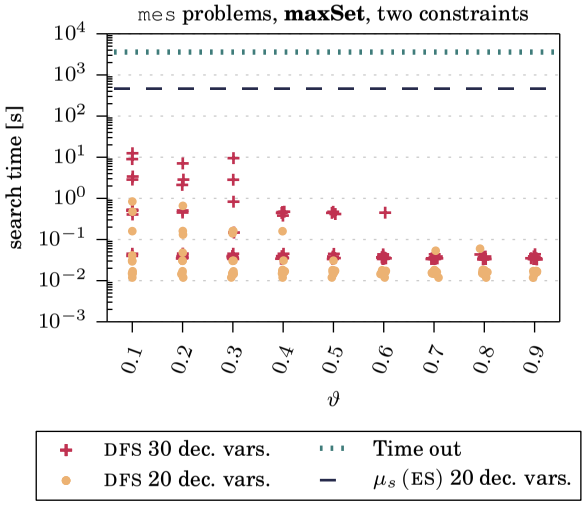
All in all, the figure does show that search times are lower for the **maxSet** setting. Most notably, we see that the search times for the `mes` problems drop significantly. Consider Figures 15b and 15e. It is remarkable that, while problems of the `mes` type seem to be hard when the constraint is tight (low values of ϑ and easy when the constraint is loose (high values of ϑ), the opposite seems to hold for problems of the `tyf` type. Analysing the data shows that for the `mes` dataset, setting all decision variables to $\top$ is a solution to each of the ten problems on twenty decision variables for $\vartheta = 0.9$, causing almost the entire search tree to be pruned away, while more search is necessary for $\vartheta = 0.1$. On the other hand: for the examples of `tyf` problems on ten decision variables, we find that for $\vartheta = 0.1$ a lot of pruning can be performed because of the strict constraints.

Some general search time observations

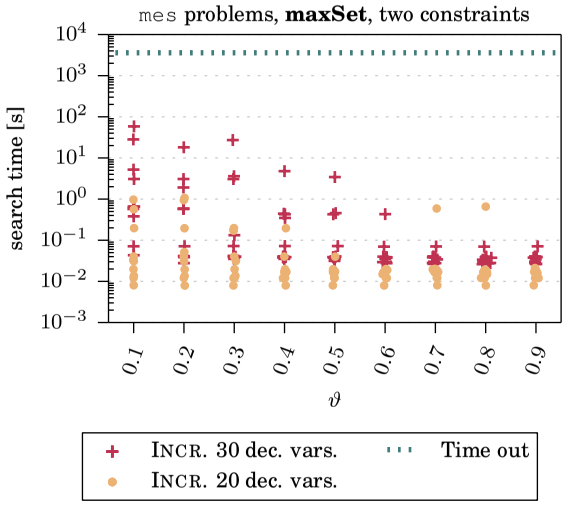
At this stage, it seems that the algorithms that build SDDs incrementally do not yet outperform the DFS-based algorithm that builds a large SDD during the preprocessing stage. After all: as Figure 13 shows, for these numbers of variables (40–90, including both probabilistic variables and decision variables), it is generally possible to compile SDDs within ten seconds. As Figures 14 and 15 show: dfs generally solves problems faster than Incremental and LazyIncremental do. Those ten seconds (at most!) needed for preprocessing are generally not sufficient for dfs to be beaten by Incremental or LazyIncremental in terms overall performance. Similar observations can be made for problems in the `fas` problem set (Figures 23 and 24 in Appendix C).

Exceptions to this rule are found in the `tyf` dataset: compiling the SDDs that represent two queries in a `tyf` problem on ten decision variables takes up to three hours, while the results in Figures 14e and 15e show that there are values for ϑ for which the Incremental search for each of the examples takes less than one hour. In these cases dfs is outperformed by Incremental.

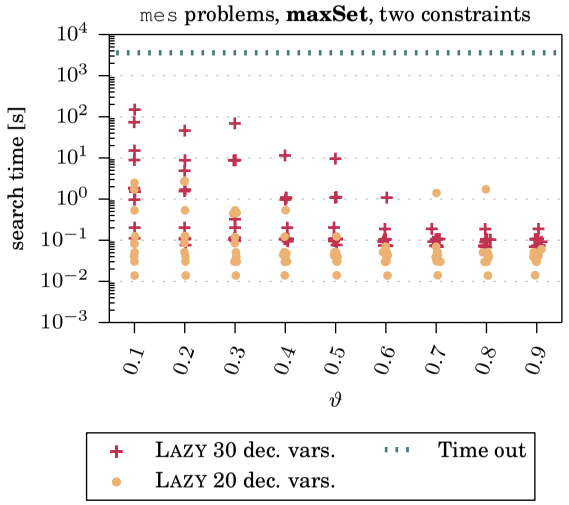
We remark that many aspects of the Incremental and LazyIncremental algorithms are still very naively implemented: such as the SDD updating method in Algorithm 4 or the order in which temporarily deterministic probabilistic variables are reset to their true probabilistic variables in Algorithm 5. Consequently, there are opportunities to make these algorithms more time-efficient. Before we continue the discussion of our experimental results by looking into the effectiveness of Incremental's and LazyIncremental's distinguishing feature, keeping the SDDs small, we first perform a small evaluation of the search efficiency of all three DFS-based methods.



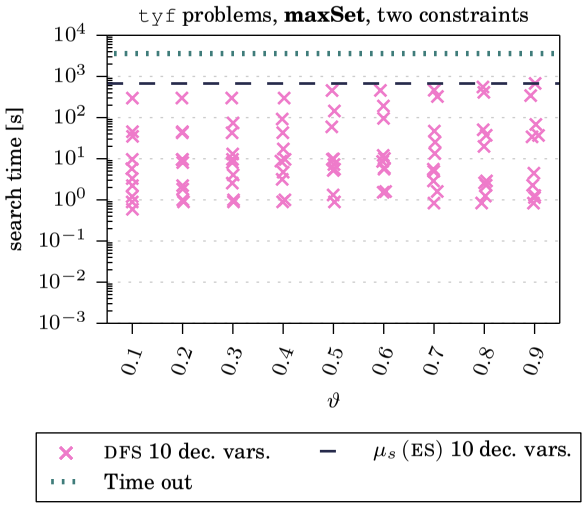
(a) ExhaustiveSearch and dfs, `mes` problem set, two constraints, μ_s (ES) $= 0.5 \pm 0.7 \cdot 10^3$ s.



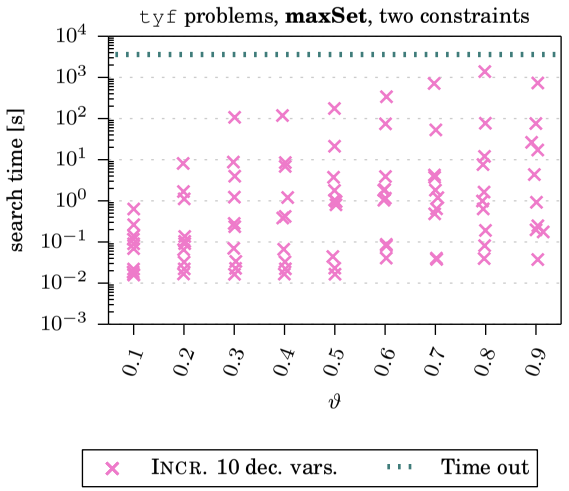
(b) Incremental, `mes` problem set, two constraints.



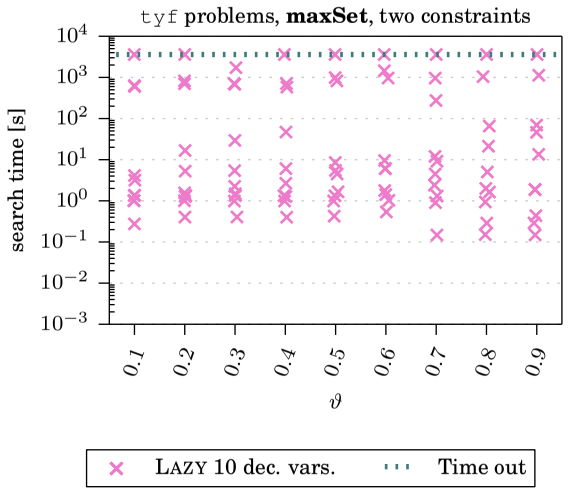
(c) LazyIncremental, `mes` problem set, two constraints.



(d) ExhaustiveSearch and dfs, `tyf` problem set, two constraints, μ_s (ES) $= 8 \pm 23 \cdot 10^3$ s (two time-outs).



(e) Incremental, `tyf` problem set, two constraints.



(f) LazyIncremental, `tyf` problem set, two constraints.

Figure 15: Search times for ExhaustiveSearch and dfs (left), Incremental (middle) and LazyIncremental (right) for `mes` problem set (top) and `tyf` problem set (bottom), in the **maxSet** setting with two constraints. Jitter added in horizontal direction to separate datapoints.

8.3 Number of search tree node visits

The results in Chapter 8.2 show that the DFS method tends to outperform the ExhaustiveSearch method in terms of search time. In Chapter 6 we discussed measures of search time complexity for the different search algorithms. The total number of evaluations of queries seems a natural choice for this complexity measure. However, using this measure, it is hard to compare the performances of all strategies. The implementation of the ProbLog Python library does not allow the evaluation of single queries (one is forced to always evaluate all), and with the removal of temporary determinism by the LazyIncremental method, counting the number of evaluations consistently also gets more complicated. We therefore use the number of visits to nodes of the search tree as a measure for the complexity in case of the DFS-based methods, and compare that to the number of strategies that are evaluated by the ExhaustiveSearch method.

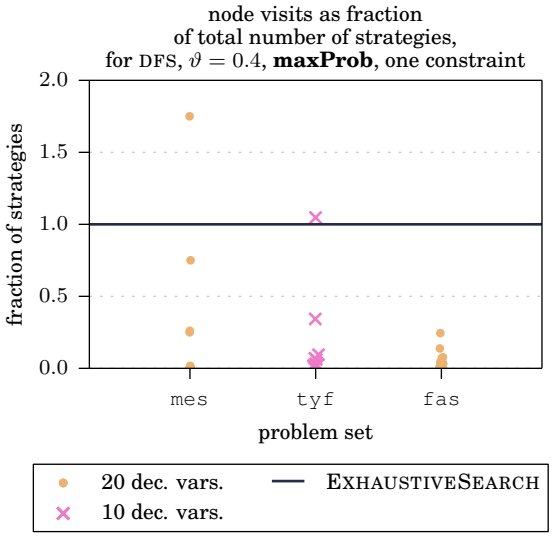


Figure 16: Number of search tree node visits of DFS (and thus Incremental and LazyIncremental methods), divided by 2^{N_d}: the number of strategies that are evaluated by the ExhaustiveSearch method.

Recall that the number of evaluations is proportional to 2^{N_d} for the ExhaustiveSearch approach. For the DFS-based algorithm (DFS, Incremental and LazyIncremental), the number of query evaluations is proportional to the number of nodes in the search tree, or 2^{N_d+1}, in the most naive implementation of a DFS-based search. This means that the pruning of the search tree must be sufficiently efficient for the DFS-based methods in order for them to generally be more efficient than the ExhaustiveSearch method in terms of query evaluations. Recall that, because of the validity check optimisation (Chapter 6.2) the worst-case number of evaluations done by DFS-based methods is also proportional to $O\left(2^{N_d}\right)$. Therefore, we know that if the number of visits to nodes in the search tree done by the DFS-based algorithms is at most 2^{N_d}, the number of query evaluations is less or equal to the number of query evaluations done by the ExhaustiveSearch method (in the worst-case limit).

Figure 16 shows for the three sets of example problems the value of $n/2^{N_d}$: the number of visits to search tree nodes (n), divided by the number of strategies for which ExhaustiveSearch needs to evaluate queries 2^{N_d}. It is shown for problems on twenty (`mes` and `fas`) and ten (`tyf`) decision variables, for problems with one constraint with $\vartheta = 0.4$. The dark line represents $2^{N_d}/2^{N_d} = 1$. All data points on or above this line correspond to problems for which DFS-based algorithms have to do as many query evaluations as the ExhaustiveSearch method, in the worst case.

The figure shows that, for these problems, pruning is in most cases sufficient to suggest that the total number of query evaluations is typically (much) smaller than the total number of strategies, and thus that DFS-based methods would typically outperform ExhaustiveSearch on these problems.

8.4 Size of SDDs

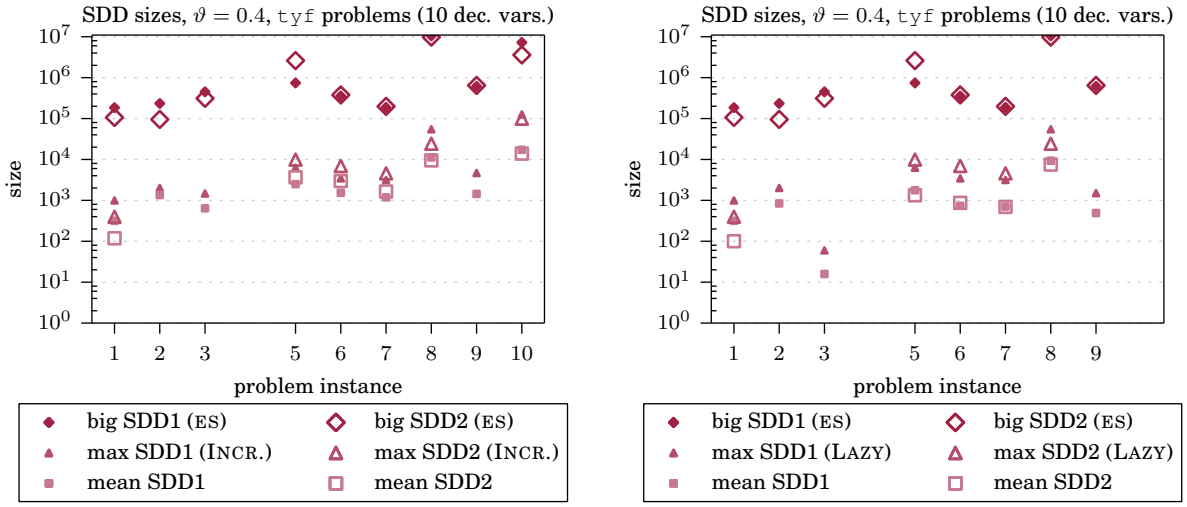
Having shown that DFS-based search algorithms tend to perform fewer query evaluations in Chapter 8.3, we move on to investigate the effectiveness of the search methods that are based on the incremental compilation (during search) of small SDDs out of probabilistic variables only: Incremental and LazyIncremental.

Mean and maximum sizes

Figures 14d and 14e show that Incremental solves problems of the `tyf` type more efficiently than DFS. As we have seen in the SDD compilation results in Figure 5, the size of the largest SDD that is compiled during the search may be a bottleneck for overall performance. We want to know if the Incremental method does indeed compile SDDs solely out of probabilistic variables that are smaller

than the big SDD built out of both decision variables and probabilistic variables that is needed for the ExhaustiveSearch and dfs methods.

Figure 17 shows for the Incremental search algorithm (Figure 17a) and the LazyIncremental algorithm (Figure 17b) what the size is of the largest SDD that is built during the entire search process, for a number of problem instances.



(a) Incremental (b) LazyIncremental

Figure 17: Sizes of SDDs built during/for different search methods, for two different queries (1, the constraint query, and 2, the optimisation query) and different instances of a `tyf` problem on ten decision variables, in the **maxProb** setting, with one constraint with $\vartheta = 0.4$. The big SDD is the SDD that is built out of both probabilistic variables and decision variables for the ExhaustiveSearch and dfs methods. The max SDD is the largest SDD that is compiled online out of solely probabilistic variables during the execution of the Incremental (a) or LazyIncremental (b) search. The figures also show the size of SDDs compiled during the execution of the Incremental and LazyIncremental incremental methods, averaged over all visited nodes of the search tree. Missing instances are those whose search process was timed out. All other sizes that are missing from the figures have value 0, and thus do not appear on a graph with a logarithmic vertical axis.

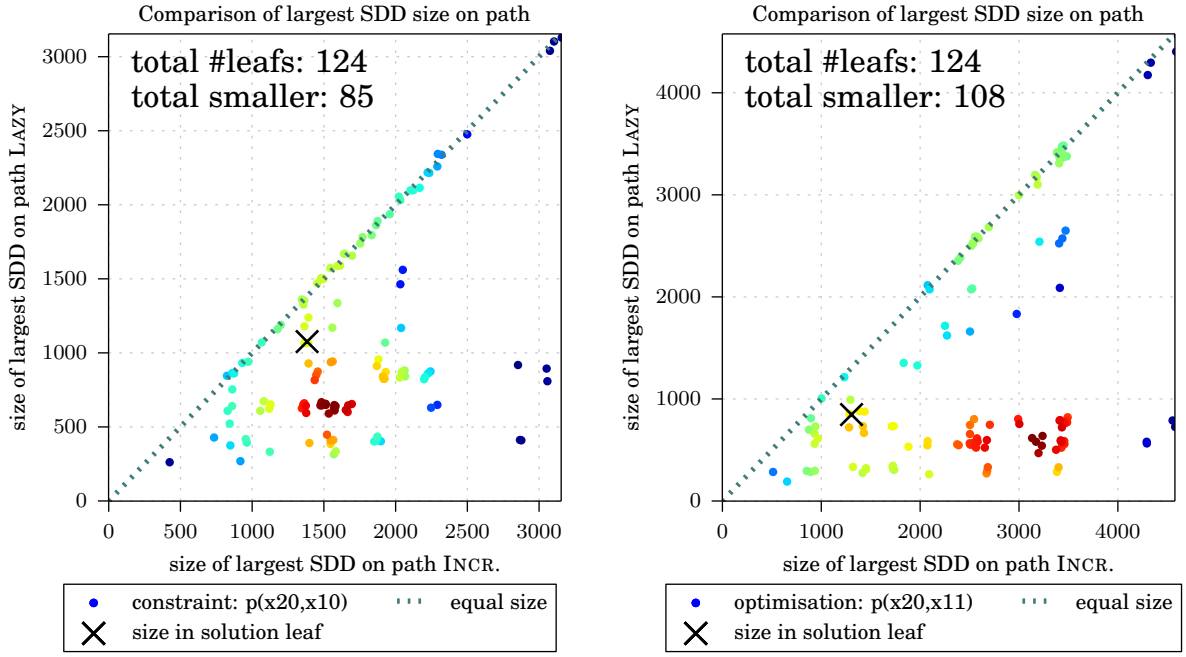
Our first observation from Figure 17 is that the big SDD compiled out of both probabilistic and decision variables for the ExhaustiveSearch and dfs methods is consistently about two orders of magnitude larger than the largest SDD that is compiled out of solely probabilistic variables during the execution of the Incremental algorithm. For problem instances 2, 3 and 9 we see that maximum size of the SDD representing the optimisation query is zero. This last result is due to the fact that for the `tyf` example problems, it is possible to have a probability of 1.0 that there is a path from the source to the target. This can be represented by an SDD of size zero.

The second observation is that the mean sizes of SDD representing the two queries are not that much smaller than the largest sizes. This is not a surprising result, since SDDs representing queries tend to get larger near the leafs of the tree, and that is also where the most nodes visits take place, so the size of the SDDs near the leafs dominates the average. Note that, as these SDDs are built incrementally, these results do not imply that in each leaf a 'large' SDD is built from scratch.

Maximum size from root to leaf

Looking at Figure 17b, we see that the SDDs built by the LazyIncremental approach do indeed tend to be smaller than the ones built by the Incremental algorithm. To further analyse this result, please consider the data presented in Figure 18.

Each data point in the figure corresponds to a leaf of the search tree that was visited during a DFS-based search for the solution to a typical example of a `tyf` problem on ten decision variables. Note that in this context: a leaf is a node in the search tree that corresponds to a *full* strategy (so one at depth N_d). I.e. internal nodes at which the search is pruned because the constraints were not satisfied are not



(a) Sizes of SDD representing the constraint query.

(b) Sizes of SDD representing the optimisation query.

Figure 18: Comparison of largest SDD encountered on each path from root to leaf, by the INCREMENTAL method (horizontal axis) and the LAZYINCREMENTAL method (vertical axis). Each point in the graphs represents one path from root to leaf. Points on the diagonal correspond to paths where the largest SDD built by the LAZYINCREMENTAL method had the same size as the largest SDD built by the INCREMENTAL method. The cross represents the leaf in which the solution to the SCOP was found. Jitter is applied to data points in order to separate them, and the colour indicates the density of the data points (red is denser than blue). Results for the constraint query (a) and the optimisation query (b) are shown for problem instance 7 of the `tyf` problems on ten decision variables (**maxProb** setting with $\vartheta = 0.4$).

included in this figure.

The figure plots the size of the largest SDD that was built along the path from the root of the search tree to that particular leaf by the INCREMENTAL method (horizontal axis) and the LAZYINCREMENTAL method (vertical axis). Recall that both methods traverse the search tree in exactly the same way and thus visit the same leafs. If the LAZYINCREMENTAL method is successful at keeping the SDDs smaller than those that are compiled by the INCREMENTAL algorithm, we expect data points below the diagonal in the plots. If the LAZYINCREMENTAL method is really good at keeping SDDs small, we expect many data points in the lower right corner of the figures.

What we observe these figures is that the LAZYINCREMENTAL method does indeed keep the largest SDDs on paths from root to leaf generally smaller than the INCREMENTAL method does. Especially for the optimisation query (Figure 18b) we see that in 108 out of 124 paths, the largest SDD that was compiled by LAZYINCREMENTAL was smaller than the largest one compiled by INCREMENTAL. Moreover, we see that even for the largest SDDs compiled by the INCREMENTAL method, some of the SDDs compiled by LAZYINCREMENTAL are an order of magnitude smaller.

From the figures we can also conclude that the SDDs built in the solution leaf were not that large compared to the largest SDDs built during the search. This important, because it means that a non-exhaustive method, based on incrementally building SDDs, that uses local search guided by smart heuristics might be a very effective one on these problems.

Appendix C contains some more examples of these figures that show slightly different behaviours. In general we can state that LAZYINCREMENTAL is most effective at keeping the SDDs small for the `tyf` examples, and less so for the `mes` and `fas` types of problems.

9 Conclusion

In this work we used probabilistic logic programming (PLP) and constraint programming (CP) to solve stochastic constraint optimisation problems (SCOPs) of a particular kind: problems that involve paths in graphs whose edges are represented by probabilistic variables and/or decision variables. These problems were encoded in ProbLog programs [13, 35]. Optimisation criteria were of the form $P(path(a, b)) \rightarrow \max$, and constraints of the form $P(path(a, c)) \leq \vartheta$, with $P(path(x, y))$ the probability of there being a path from x to y.

Naive methods for solving these types of problems use an initialisation phase to compile optimisation and constraint queries to large sentential decision diagrams (SDDs) consisting of both probabilistic and decision variables. After this initialisation phase, they perform a search (either based on enumeration for the ExhaustiveSearch method or depth-first search (DFS) for the dfs method) over assignments to decision variables to find the optimal strategy.

Because the compilation of such large SDDs is infeasible for all but the smallest problems, we proposed two incremental methods that build smaller SDDs during the search process (and do not require the building of SDDs during a preprocessing phase). The Incremental method builds SDDs out of probabilistic variables only, in a manner that is reminiscent of $T_{\mathcal{P}}$-compilation [39]. The SDDs that are built by Incremental are kept smaller because they contain probabilistic variables only, and their incremental compilation is stopped once it becomes clear that constraints are being violated. The LazyIncremental method takes this concept a step further, by temporarily assuming that probabilistic variables are in fact deterministically 'true', thus eliminating even more variables from the SDDs, and only reviewing this assumption when constraints are being violated or an exact probability for the optimisation query is required.

We have tested these methods on small example problems consisting of social networks with different topologies and various types of probabilistic and decision variables. From these empirical results, we conclude that the Incremental method outperforms the naive ExhaustiveSearch and dfs methods on problems with certain characteristics. Specifically: problems for which it is possible to obtain proofs for paths from a source node to a target node by assigning the value $\top$ ('true') to only one decision variable (these are the `tell-your-friends`, or `tyf`, problems described in Chapter 5). For problem types where sets of decision variables have to have value $\top$, Incremental is outperformed by dfs. While the LazyIncremental method proves to be effective in keeping SDDs smaller than the SDDs generated by Incremental, the overhead is too large for LazyIncremental to outperform Incremental on any of the problems that were tested, in terms of search time.

We observe that the SDDs that represent the queries for the strategy that is the optimal solution to the problems, are typically kept relatively small by both the Incremental and the LazyIncremental methods. This indicates that, given the right heuristics and strategy, these incremental methods are suitable for the development of greedy approximation algorithms.

Taking into consideration that the implementation for Incremental and LazyIncremental is very naive, but they are effective in keeping SDDs small, we conclude that these methods have the potential to be further developed into algorithms that outperform the naive methods on more different types of examples. We expand on this in the next chapter.

10 Future work

As the empirical results presented in Chapter 8 indicate, the INCREMENTAL and LAZYINCREMENTAL methods (have the potential to) outperform the naive EXHAUSTIVESEARCH and DFS methods on certain types of SCOPs. Since these algorithms are still implemented rather naively, we propose the following lines of enquiry for improving the efficiency of these methods:

1. More efficient updating of the SDD manager: no need to propagate changes to ancestors of a DAG node if the corresponding formula has not changed its value.
2. Using heuristics to determine an order in which decision variables are assigned values, because the order makes a difference for the number of search tree nodes that are visited and the size of the SDDs that are compiled in those nodes.
3. Making Simple subset pruning (SSSP) smarter.
4. Assigning truth values to groups of decision variables rather than individual variables.
5. Using heuristics to determine an order in which constraints are evaluated.
6. Using efficiently computable upper bounds for the optimisation query and/or efficiently computable lower bounds for constraints to detect pruning opportunities more efficiently (for the **maxProb** setting).
7. More efficient (partial) removal of temporary determinism from SDDs by taking into consideration which probabilistic variables might actually contribute to the total probability of a query (based on truth values of decision variable).
8. Developing better heuristics for determining the order in which temporarily deterministic variables are reviewed.
9. Improving the order in which temporary determinism is removed from constraints.
10. Investigating how general inference techniques developed by the CP community can be used for the development of more specific inference techniques for the kind of SCOPs studied in this work.
11. Implementing lifted inference techniques [5, 36] to improve efficiency.

All of these are improvements of the efficiency of the incremental algorithms presented in this work. We also see opportunities for the development of variations on these methods:

1. Developing support for multi-optimisation problems.
2. Developing support for different optimisation settings (**minProb** and **minSet**).
3. Developing support for programs that use negation.
4. Developing support for programs that use (possibly negative) utilities next to probabilities.
5. Using local search or greedy search for approximate algorithms.

Finally, the integration of the developed incremental solving techniques with existing PLP languages and CP solvers would provide availability of efficient implementations of the incremental algorithms for solving SCOPs to anyone.

11 Acknowledgements

This work could not have been done without the support of professor Luc De Raedt, who welcomed me in the DTAI group (*Declaratieve Talen en Artificiele Intelligentie*, or Declarative Languages and Artificial Intelligence) at the Computer Science department of KU Leuven. I am grateful to him and to dr. Anton Dries and dr. Angelika Kimmig for supervising me during the time I spent in Leuven.

I would also like to thank the other members of the department in general and the DTAI group in particular for their hospitality. They have welcomed me more warmly into the group than I could ever have expected.

This work was also supported by the Erasmus+ programme of the European Commission, which awarded me a grant for my stay in Belgium.

At Leiden University I would like to thank dr. Jeannette de Graaf, dr. Marcello Bonsangue and professor Aske Plaat. Their support during my time as a student at the Leiden Institute of Advanced Computer Science (LIACS) was of great importance to me. In the same spirit, I would like to thank dr. Frank Takes and professor Thomas Bäck for sparking my interest in the field of computer science in the first place, and encouraging me to continue to give it my one hundred percent later on.

I would like to thank professor Peter Lucas and dr. Marcello Bonsangue for being my supervisors for this project, and for their feedback and support.

Finally, I am very grateful to dr. Siegfried Nijssen. His trust, dedication, understanding, humour, preciseness, creativity, sincerity and occasional kick-in-the-butt have been incredibly helpful during the entire project.

Anna Latour, Leiden, December 21st, 2016

References

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '93. Santa Clara, California, USA: IEEE Computer Society Press, 1993, pp. 188–191. ISBN: 0-8186-4490-7.

[2] Chitta Baral and Matt Hunsaker. "Using the Probabilistic Logic Programming Language P-log for Causal and Counterfactual Reasoning and Non-Naive Conditioning." In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*. IJCAI'07. 2007, pp. 243–249.

[3] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. 3rd. Springer Publishing Company, Incorporated, 2012. ISBN: 1447141288, 9781447141280.

[4] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* 2006. URL: `http://www.learnprolognow.org` (visited on 04/2016).

[5] Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. "Lifted first-order probabilistic inference". In: *Proceedings of the 19th international joint conference on Artificial intelligence*. Citeseer. 2005, pp. 1319–1325.

[6] Abraham Charnes and William W. Cooper. "Chance-Constrained Programming". In: *Management Science* 6.1 (1959), pp. 73–79.

[7] Mark Chavira and Adnan Darwiche. "On probabilistic inference by weighted model counting". In: *Artificial Intelligence* (2008).

[8] Arthur Choi and Adnan Darwiche. *SDD Advanced-User Manual*. English. Version 1.1. Automated Reasoning Group, Computer Science Department of University of California, Los Angeles. Nov. 2013. 32 pp. November 7, 2013.

[9] Alain Colmerauer and Philippe Roussel. "The birth of Prolog". In: *History of programming languages—II*. ACM. 1996, pp. 331–367.

[10] Adnan Darwiche. "SDD: A new canonical representation of propositional knowledge bases". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*. Vol. 2. IJCAI'11 1. 2011, pp. 819–826.

[11] Adnan Darwiche and Pierre Marquis. "A Knowledge Compilation Map". In: *J. Artif. Int. Res.* 17.1 (Sept. 2002), pp. 229–264. ISSN: 1076-9757.

[12] Luc De Raedt and Angelika Kimmig. "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1 (2015), pp. 5–47.

[13] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery." In: *Proceedings of the Twentieth International Joint Conference on Artifical Intelligence*. Vol. 7. IJCAI'07. 2007, pp. 2462–2467.

[14] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. "Statistical Relational Artificial Intelligence: Logic, Probability, and Computation". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 10.2 (2016), pp. 1–189.

[15] DTAI Research Group. *ProbLog 2.1 manual*. English. KU Leuven. 2016.

[16] KU Leuven DTAI Research Group. *ProbLog Python library*. `https://bitbucket.org/problog/problog`. Version 2.1. 2015–2016.

[17] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. "Inference and learning in probabilistic logic programs using weighted Boolean formulas". In: *Theory and Practice of Logic Programming* 15 (03 May 2015), pp. 358–401. ISSN: 1475-3081.

[18] Solomon W. Golomb and Leonard D. Baumert. "Backtrack programming". In: *Journal of the ACM (JACM)* 12.4 (1965), pp. 516–524.

[19] KU Leuven DTAI Research Group. *ProbLog tutorial: Social networks (Friends & Smokers)*. `https://dtai.cs.kuleuven.be/problog/tutorial/basic/05_smokers.html`. 2015.

[20] Kristian Kersting and Luc De Raedt. "Basic principles of learning Bayesian logic programs". In: *Probabilistic Inductive Logic Programming*. Springer, 2008, pp. 189–221.

[21] Alan K. Mackworth and Eugene C. Freuder. "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems". In: *Artificial intelligence* 25.1 (1985), pp. 65–74.

[22] Knot Pipatsrisawat and Adnan Darwiche. "A Lower Bound on the Size of Decomposable Negation Normal Form". In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI'10. Atlanta, Georgia: AAAI Press, 2010, pp. 345–350.

[23] David Poole. "Representing Diagnostic Knowledge for Probabilistic Horn Abduction". In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*. Vol. 2. IJCAI'91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 1129–1135.

[24] David Poole. "The Independent Choice Logic and Beyond". In: *Probabilistic Inductive Logic Programming: Theory and Applications*. Springer Berlin Heidelberg, 2008, pp. 222–243.

[25] David Poole. "The independent choice logic for modelling multiple agents under uncertainty". In: *Artificial Intelligence* 94.1 (1997), pp. 7 –56.

[26] Matthew Richardson and Pedro Domingos. "Markov Logic Networks". In: *Mach. Learn.* 62.1-2 (Feb. 2006), pp. 107–136.

[27] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[28] Dan Roth. "On the Hardness of Approximate Reasoning". In: vol. 82. 1-2. Essex, UK: Elsevier Science Publishers Ltd., 1996, pp. 273–302.

[29] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. 3rd ed. Prentice Hall, Dec. 2009.

[30] Nikolaos V. Sahinidis. "Optimization under uncertainty: state-of-the-art and opportunities". In: *Computers & Chemical Engineering* 28.6–7 (2004). FOCAPO 2003 Special issue, pp. 971–983.

[31] Taisuke Sato. "A Statistical Learning Method for Logic Programs with Distribution Semantics". In: *IN PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING (ICLP95)*. MIT Press, 1995, pp. 715–729.

[32] Taisuke Sato and Yoshitaka Kameya. "Parameter learning of logic programs for symbolic-statistical modeling". In: *Journal of Artificial Intelligence Research* 15 (2001), pp. 391–454.

[33] Taisuke Sato, Neng-Fa Zhou, Yoshitaka Kameya, and Yusuke Izumi. *PRISM*. `http://rjida.meijo-u.ac.jp/prism`. Version 2.2. 2016.

[34] Afany Software. *B-Prolog*. `www.picat-lang.org/bprolog`. Version 8.1. 2014.

[35] Guy Van den Broeck, Ingo Thon, Martijn Van Otterlo, and Luc De Raedt. "DTProbLog: A decision-theoretic probabilistic Prolog". In: *Proceedings of the twenty-fourth AAAI conference on artificial intelligence*. AAAI Press. 2010, pp. 1217–1222.

[36] Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. "Lifted probabilistic inference by first-order knowledge compilation". In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*. AAAI Press. 2011, pp. 2178–2185.

[37] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. "Logic programs with annotated disjunctions". In: *International Conference on Logic Programming*. Springer. 2004, pp. 431–445.

[38] Fabien Viger and Matthieu Latapy. "Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence". In: *Computing and Combinatorics: 11th Annual International Conference, COCOON 2005 Kunming, China, August 16–19, 2005 Proceedings*. Ed. by Lusheng Wang. Springer Berlin Heidelberg, 2005, pp. 440–449.

[39] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. "Anytime inference in probabilistic logic programs with Tp-compilation". In: IJCAI'15 (2015).

[40] Toby Walsh. "Stochastic constraint programming". In: *ECAI*. Vol. 2. 2002, pp. 111–115.

List of Symbols

$\mathbf{A}$	SDD array: contains for each node in a DAG a pointer to the root of an SDD in the SDD manager.
$\mathbb{B}$	Set of Booleans: $\{\top, \bot\}$.
C	Choice mapping *some* decision variables d in $\mathscr{D}_{\mathscr{F}}$ to truth values: $\{(d \mapsto b) \mid b \in \mathbb{B}\}$.
$\mathscr{C}$	Set of (f, ϑ) pairs (generally) representing constraints of the form $P(f) \leq \vartheta$.
$\mathscr{D}$	Set of decision clauses or decision variables in a logic program.
$\mathscr{D}_{\mathscr{F}}$	Set of decision clauses or decision variables in a logic program that are relevant to formulas in $\mathscr{F}$.
d	Decision variable.
δ	Depth in search tree (root has depth 0).
$\mathcal{DT}$	Decision theoretic probabilistic logic program.
$\bot$	Truth value 'False'.
$\mathscr{F}$	Set of logic formulas.
f_c	Logic formula corresponding to a constraint query q_c. Depending on context, f_c may also refer to the (root of the) compiled SDD or DAG that represents q_c.
f_o	Logic formula corresponding to an optimisation query q_o. Depending on context, f_o may also refer to the (root of the) compiled SDD or DAG that represents q_o.
$\mathscr{I}$	Interpretation of a set of logic formulas $\mathscr{F}$, mapping atoms in $\mathscr{F}$ to truth values: $\{a \mapsto b \mid a \in f, f \in \mathscr{F}, b \in \mathbb{B}\}$.
L	Logic program $\{c_1, \ldots, c_n\}$, with c_i a clause.
L_T	Set of all logic programs L that can be sampled from probabilistic logic program T.
μ_s	Mean search time.
N_c	Number of constraints.
N_d	Number of relevant decision variables: $\lvert\mathscr{D}_{\mathscr{F}}\rvert$.
N_o	Number of optimisation criteria.
N_p	Number of relevant probabilistic variables: $\lvert\mathscr{P}_{\mathscr{F}}\rvert$.
N_q	Total number of queries: $N_o + N_c$.
N_v	Total number of probabilistic and decision variables: $N_p + N_d$.
$\mathscr{P}$	Set of probabilistic clauses or probabilistic variables in a logic program.
$\mathscr{P}_{\mathscr{F}}$	Set of probabilistic clauses or probabilistic variables in a probabilistic logic program that are relevant to formulas in $\mathscr{F}$.
Q	Queue of nodes in a DAG that need to be updated during the execution of the Incremental or LazyIncremental method.
q	Query to (probabilistic) logic program.
S	Set of strategies σ.
σ	Strategy mapping decision variables to truth values: $\{(d \mapsto b) \mid d \in \mathscr{D}, b \in \mathbb{B}\}$. For shortness sometimes given as set of all decision variables that map to $\top$.
$\sigma_\bot$	Set of all decision variables in strategy σ that map to $\bot$: $\{d \mid (d \mapsto \bot) \in \sigma\}$.
σ_s	Standard deviation of search time.
$\sigma_\top$	Set of all decision variables in strategy σ that map to $\top$: $\{d \mid (d \mapsto \top) \in \sigma\}$.
$\mathbf{M}$	SDD manager: an object containing several (sub) SDDs that represent formulas.

T	Probabilistic logic program $\{(p_1, c_1), \ldots, (p_n, c_n)\}$ with p_i the probability of clause c_i.
$\top$	Truth value 'True'.
θ	Substitution $\{x_1/t_1, \ldots, x_n/t_n\}$.
ϑ	Threshold for probabilistic constraint.
t_s	Search time.

List of Acronyms

SSSP	Simple subset pruning.
`fas`	`friends-and-smokers`.
`mes`	`messages`.
`tyf`	`tell-your-friends`.
ADD	Algebraic Decision Diagram.
CCOP	Chance-Constrained Optimisation Problem.
COP	Constraint Optimisation Problem.
CP	Constraint Programming.
CSP	Constraint Satisfaction Problem.
DAG	(AND/OR) Directed Acyclic Graph.
DFS	Depth-First Search.
DNF	Disjunctive Normal Form.
ICL	Independent Choice Logic.
KB	Knowledge Base.
MLN	Markov Logic Network.
OBDD	Ordered Binary Decision Diagram.
PLP	Probabilistic Logic Programming.
SCOP	Stochastic Constraint Optimisation Problem.
SDD	Sentential Decision Diagram.
WMC	Weighted Model Count.

A An introduction to logic

In this chapter we give a basic introduction in *propositional logic* (Boolean operators and atoms) and *first-order logic* (reasoning with Boolean operators, quantifiers, relations, predicates and *sets* of atoms). This chapter is based on *Mathematical Logic for Computer Science*, Mordechai Ben-Ari [3]. Here we introduce some of the logical concepts that are relevant for this work. For a detailed discussion on the exact logical rules or proofs of theorems, consult for example Ben-Ari's work.

This chapter serves mainly as a reference for some of the concepts that are used throughout the rest of this work, and can be consulted if short descriptions in the list of symbols are not sufficient.

A.1 Propositional logic

We introduce propositional logic formulas, interpretations, models and truth tables.

Propositional logic formulas

A *formula* in propositional logic consists of *atomic propositions*, or *atoms*, and *Boolean operators*. Atoms are typically written as lowercase letters, such as p, q and r. The two constants, `True` and `False`, are also atoms and written as $\top$ and $\bot$, respectively. We denote the set of all atomic propositions as $\mathcal{P}$. Atomic propositions can be assigned a meaning. For example: we might assign the meaning 'Spock is a Vulcan' to the symbol p, 'Spock is a Borg' to q and 'Spock acts rationally' to r. We can combine the atomic propositions to formulas using the Boolean operators given in Table 4.

Table 4: Boolean operators in propositional logic, with their type and order of precedence (lower means stronger binding).

Operator	Symbol	Type	Order
negation	$\neg$	unary	1
conjunction	$\wedge$	binary	2
disjunction	$\vee$	binary	3
nand	$\uparrow$	binary	2
nor	$\downarrow$	binary	3
implication	$\rightarrow$	binary	4
equivalence	$\leftrightarrow$	binary	5
exclusive or	$\oplus$	binary	5

Boolean operators can be applied to both atoms and formulas. For example, we can conjoin atoms p and q to $p \vee q$, and then create an implication: $(p \vee q) \rightarrow r$ (which may also be written as $p \wedge q \rightarrow r$, because of the order of precedence of Boolean operators). With the Vulcan example above, this has the meaning:

$$\text{Spock is a Vulcan} \ \vee \ \text{Spock is a Borg} \ \rightarrow \ \text{Spock acts rationally},$$

which is read as 'If Spock is a Vulcan or Spock is a Borg, then Spock acts rationally.' Note that this is a *declarative sentence*: a sentence that is either true or false, and thus different from a sentence like 'Who is this Spock person, anyway?' We use propositional logic for constructing declarative sentence, or formulas. We denote the set of all logical formulas that can be constructed using the atomic propositions in $\mathcal{P}$ and the Boolean operators in Table 4 with $\mathcal{F}$.

Note that the first three operators in Table 4 are fundamental, and all the other ones can be expressed in terms of the first three. For example, we can write $a \rightarrow b$ as $\neg a \vee b$. We say that these two formulas are *logically equivalent*, which means that the one can be substituted for the other without changing the meaning of what was written. Logical equivalence is denoted with $\equiv$. In our example: $a \rightarrow b \equiv \neg a \vee b$.

Interpretations of (sets of) formulas

We can use a set of formulas $S = \{A_1, \ldots\}$ to describe a world for which exactly that set of formulas holds. An *interpretation* $\mathscr{I}_S$ of S maps each atom in the union of sets of atoms of formulas in S to a truth value in $\{\top, \bot\} = \mathbb{B}$. Now for any $A_i \in S$, $v_{\mathscr{I}_S}(A_i)$ represents the *truth value* of A_i *under* S.

Suppose we have set of formulas $S = \{p, q, r, p \vee q, p \vee q \rightarrow r\}$ such that $S \subseteq \mathcal{F}$, with interpretation

$\mathscr{I}_S(p) = \top$, $\mathscr{I}_S(q) = \bot$ and $\mathscr{I}_S(r) = \top$, then the truth values of the formulas under $\mathscr{I}_S$ are:

$$\begin{aligned} v_{\mathscr{I}_S}(p) &= \mathscr{I}_S(p) = \top \\ v_{\mathscr{I}_S}(q) &= \mathscr{I}_S(q) = \bot \\ v_{\mathscr{I}_S}(r) &= \mathscr{I}_S(r) = \top \\ v_{\mathscr{I}_S}(p \vee q) &= \top \\ v_{\mathscr{I}_S}(p \vee q \rightarrow r) &= \top \end{aligned}$$

Note that the *equivalence operator* and *logical equivalence* are related, but different. Particularly, it holds that $A_1 \equiv A_2$ if and only if $A_1 \leftrightarrow A_2$ *is true in every interpretation*, for $A_1, A_2 \in \mathcal{F}$.

Models and truth tables

If, for a certain interpretation $\mathscr{I}_S$, it holds that $v_{\mathscr{I}_S}(A_i) = \top$ for each $A_i \in S$, the interpretation is called a *model* of S. If there exists a model for a set of propositional logic formulas S, we call this set *satisfiable*. An example of a satisfiable set of propositional logic formulas is $\{a, b\}$. If all interpretations for S are models of S, we say that S is *valid* or that S is a *tautology*. An example of a valid set of formulas is $\{a \vee \neg b\}$. We denote the fact that S is valid as $\models S$. If a set of formulas has no models, that set is called *unsatisfiable*. An example of an unsatisfiable set is $\{a, \neg b\}$. Finally, if there exist interpretations for which not all logical formulas in S have truth value $\top$, S is *falsifiable*, denoted $\not\models S$. An example of a falsifiable set of formulas is $\{a, b\}$. These terms are also used for individual formulas, rather than sets of formulas.

Let $S \subseteq \mathcal{F}$ be a set of formulas and $A \in \mathcal{F}$ a particular formula. If every model for S is also a model for A, we say that A is a *logical consequence* of S, denoted $S \models A$.

We can analyse the models of (sets of) formulas by creating truth tables, such as the one shown in Table 5. In this table, we write down each possible interpretation for the atoms in $\{p, q, r\}$, and compute the truth value for the formula $p \wedge q \rightarrow r$.

Table 5: A truth table for $p \vee q \rightarrow r$.

p	q	r	$p \vee q$	$p \vee q \rightarrow r$
$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$

A.2 First-order logic

We now continue with a short introduction to *first-order logic*, a generalisation of propositional logic. We will start by explaining this generalisation, then discuss the concept of interpretation for first-order logic, and finally have some notes on the subject of substitution.

Functions in formulas

Recall the example about Spock and acting rationally. Spock is not the only member of the Star Trek cast we might want to consider: Tuvok is another character who might act rationally. Suppose we have a Star Trek set of propositional logic sentences that looks like this:

$$\{\text{Spock is a Vulcan } \vee \text{ Spock is a Borg } \rightarrow \text{ Spock acts rationally}\}.$$

Now we want to expand this set by adding information about Tuvok, so we obtain:

$$\begin{aligned} \{&\text{Spock is a Vulcan } \vee \text{ Spock is a Borg } \rightarrow \text{ Spock acts rationally}, \\ &\text{Tuvok is a Vulcan } \vee \text{ Tuvok is a Borg } \rightarrow \text{ Tuvok acts rationally}\}. \end{aligned}$$

Now we want to include other information about Vulcans and Borgs, which yields the following set:

$$\begin{aligned} \{&\text{Spock is a Vulcan } \vee \text{ Spock is a Borg } \rightarrow \text{ Spock acts rationally}, \\ &\text{Tuvok is a Vulcan } \vee \text{ Tuvok is a Borg } \rightarrow \text{ Tuvok acts rationally}, \\ &\text{Vulcans can do mind-melds } \wedge \text{ Spock is a Vulcan } \rightarrow \text{ Spock can do mind-melds}, \\ &\text{Vulcans can do mind-melds } \wedge \text{ Tuvok is a Vulcan } \rightarrow \text{ Tuvok can do mind-melds}, \\ &\text{Borgs can assimilate } \wedge \text{ Spock is a Borg } \rightarrow \text{ Spock can assimilate}, \\ &\text{Borgs can assimilate } \wedge \text{ Tuvok is a Borg } \rightarrow \text{ Tuvok can assimilate}\} \end{aligned} \tag{20}$$

Our Star Trek set of propositional formulas will expand rapidly any time we add new information about particular characters, or about Vulcans or Borgs in general to our set. We can encode the same information more efficiently using *first-order logic*.

Where we needed the sentences 'Spock is a Vulcan' and 'Tuvok is a Vulcan' to express the facts that Spock and Tuvok are Vulcans in propositional logic, we use a *predicate* to express the same information in first-order logic:

$$\begin{aligned}&\text{isa(Spock, Vulcan)},\\&\text{isa(Tuvok, Vulcan)}.\end{aligned}$$

A predicate is a function that maps an n-ary domain (in this case the binary domain of characters from the Star Trek cast and humanoids from the Star Trek universe) to truth values. In this case, isa(Spock, Vulcan) maps to $\top$ because Spock is a Vulcan, whereas isa(Picard, Vulcan) would map to $\bot$, because captain Picard is not.

The improvement in compactness comes from the fact that we can now use variables to express rules, which enables us to write down the rules of (20) more compactly as:

$$\begin{aligned}\{&\text{isa(X, Vulcan)} \lor \text{isa(X, Borg)} \rightarrow \text{actsRationally(X)},\\&\text{canDoMindMelds(Vulcan)} \land \text{isa(X, Vulcan)} \rightarrow \text{canDoMindMelds(X)}\ ,\\&\text{canAssimilate(Borg)} \land \text{isa(X, Borg)} \rightarrow \text{canAssimilate(X)}\ .\}\end{aligned}$$

Herbrand interpretations

Note that the use of function in formulas makes it harder to define the set of all possible interpretations for those formulas. This is partly due to the fact that the *domain* of the interpretation is not immediately clear.

Where an interpretation in propositional logic simply takes the set of constants (such as q or 'Spock is a Vulcan') in a formula as its domain, first-order logic needs the concept of an *Herbrand universe*. The Herbrand universe H_S of a set of formulas S is constructed recursively and contains:

- each constant $a \in \mathcal{A}$, with $\mathcal{A}$ the set of all constants that occur in formulas in S;
- each function symbol $f \in \mathscr{F}$, with $\mathscr{F}$ the set of all function symbols that occur in formulas in S;
- all terms that can be derived by recursively applying functions to elements in the Herbrand universe.

In other words: a Herbrand universe is the set of all *ground* atoms that can be generated recursively using the predicates, functions and constants in S. Here a formula is called 'ground' if it contains no variables.

Now the *Herbrand interpretation* of a set of formulas is defined as follows:

$$\mathscr{I} = \{H_S, \{R_1, \ldots, R_k\}, \{f_1, \ldots, f_\ell\}, \mathcal{A}\},$$

with $\{R_1, \ldots, R_k\}$ arbitrary relations over the domain H_S and $\{f_1, \ldots, f_\ell\}$ functions over variables in domain H_S, such that assignments in $\mathscr{I}$ are as follows:

$$\begin{aligned}v_{\mathscr{I}}(a) &= a,\\v_{\mathscr{I}}\left(f(t_1, \ldots, t_n)\right) &= f\left(v_{\mathscr{I}}(t_1), \ldots, v_{\mathscr{I}}(t_n)\right).\end{aligned}$$

In other words: in an Herbrand interpretation, constant symbols are interpreted as the constants they represent, and function symbols are interpreted as the functions they represent. Note that an interpretation only contains ground expressions.

Substitutions

In this work, we use the concept of *substitution* in different contexts. For example, we use it in the explanation of SLD-resolution in Chapter 3.2 when variables are substituted for constants. We also use it in Chapters 6.3 and 6.4 to describe the formulas for which the IncrementalAlgorithm and LazyIncrementalAlgorithm algorithms build SDDs.

A substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ maps each variable x_i to a term t_i that is not identical to x_i. When applied to an expression f, which is written as $f\theta$, all x_i that occur in f are *instantaneously* replaced by their corresponding t_i. This instantaneous substitution means that if t_i is identical to some x_j in θ, x_i is not replaced by t_j, but simply by t_i.

B An introduction to knowledge compilation

Knowledge compilation is a way of handling computational intractability in propositional reasoning. The computational complexity is concentrated in an off-line phase in which propositional formulas are compiled into a target language that allows for polytime querying in the on-line phase of the problem solving. [11]

In this chapter we provide an introduction to *ordered binary decision diagrams* (OBDDs) and *sentential decision diagrams* (SDDs), but first we discuss a more naive way of visualising propositional formulas: *AND/OR directed acyclic graphs* (DAGs).

B.1 AND/OR DAGs

Consider the directed probabilistic graph and its (decision theoretic) probabilistic logic program in Figure 19, and consider all the simple paths (no loops) from a to b. These can be expressed in the following logical formula:

$$\begin{aligned} p_{ab} = &[d_{ab} \wedge e_{ab}] \vee \\ &([d_{ac} \wedge e_{ac}] \wedge [d_{cb} \wedge e_{cb}]) \vee \\ &([d_{ad} \wedge e_{ad}] \wedge ([d_{db} \wedge e_{db}] \vee ([d_{dc} \wedge e_{dc}] \wedge [d_{cb} \vee e_{cb}]))) , \end{aligned} \tag{21}$$

where p_{ab} is a Boolean variable indicating if there is a path from a to b, d_{uv} is a Boolean (decision) variable indicating if the edge (u, v) is 'allowed' to exist, and e_{uv} is a (probabilistic) Boolean variable corresponding to the `edge()` predicate in Figure 19 that is true with a certain probability.

```
?::d(a,b).
?::d(b,c).
?::d(a,c).
?::d(a,d).
?::d(c,b).
?::d(d,b).
?::d(d,c).

0.4::edge(a,b).
0.8::edge(b,c).
0.5::edge(a,c).
0.9::edge(a,d).
0.7::edge(c,b).
0.3::edge(d,b).
0.2::edge(d,c).

connection(X,Y) :- edge(X,Y), d(X,Y).
path(X,Y) :- connection(X,Y).
path(X,Y) :- connection(X,Z), path(Z,Y).

query(path(a,b)).
query(path(a,c)).
```

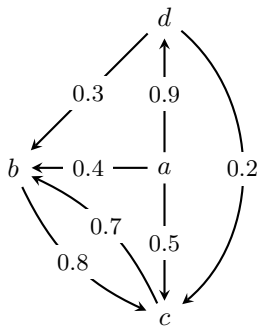


Figure 19: A program representing a network in which each query is represented by a probabilistic variable and a decision variable. Queries to the existence of at least one path from a to b and at least one from a to c are included.

Note that some of the Boolean variables occur more than once in this formula, so the formula could be represented more compactly. A way to do that, is to compile the formula into an AND/OR graph, an example of which is shown in Figure 20.

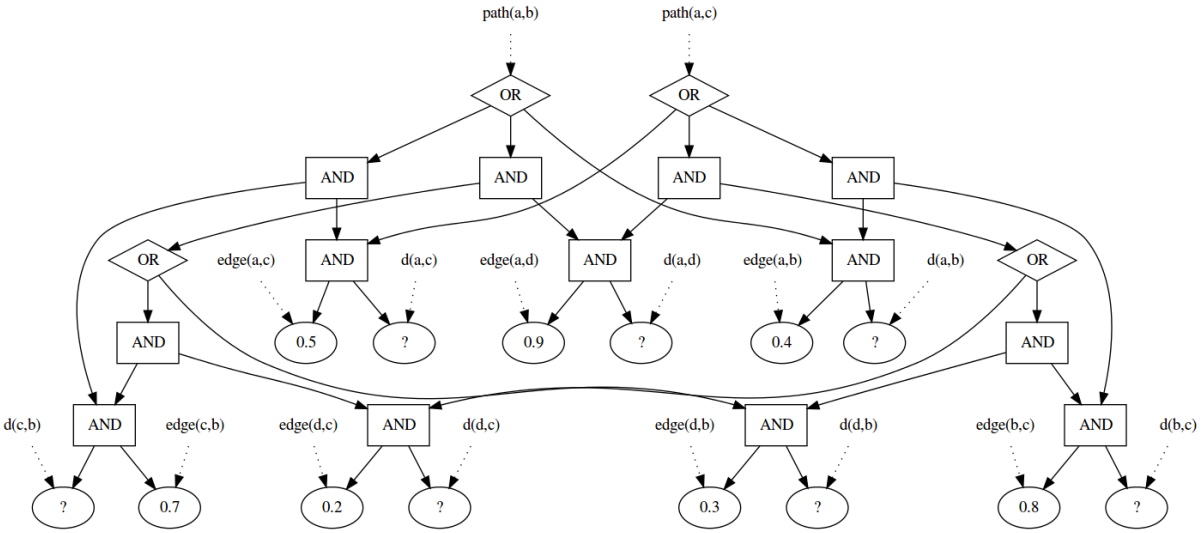


Figure 20: The AND/OR DAG corresponding to the queries in the program of Figure 19.

In fact Figure 20 shows the AND/OR DAGs for both queries in the program of Figure 19. It is simply a visualisation of Boolean logic formulas.

The *size* of an AND/OR DAG is defined as the number of its nodes. Thus, the DAG shown in Figure 20 has size 31.

Note that in the particular implementation of ProbLog used for this work, the DAGs get slightly larger due to the fact that cycle breaking to keep paths simple happens in the compilation phase rather than in the ProbLog program itself. This choice was made because the compilation of ProbLog programs that only allow simple paths is extremely slow due to the usage of Prolog lists.

B.2 Ordered Binary Decision Diagrams

Recall the truth table in Table 5, Appendix A.1. Note that, in order to list all possible interpretations explicitly, the number of rows doubles with each addition of a new variable. A more compact and intuitive way of encoding the same information is to convert it to a *decision diagram*. In a decision diagram each node corresponds to an atom and each branch corresponds to either choosing that atom to be true, or choosing it to be false. The leaves of the tree correspond to the truth value of the formula under the interpretation that is obtained by following the choices made in the path from root to leaf.

An example is shown in Figure 21a. Decision trees can grow quite large, as the total number of nodes in the tree is $\sum_{i=0}^{n} 2^i$, with n the number of variables in the formula. We can represent the same information more space efficiently by converting the decision tree in an OBDD. An example of the algorithm for turning a decision tree into an OBDD is shown in Figure 21.

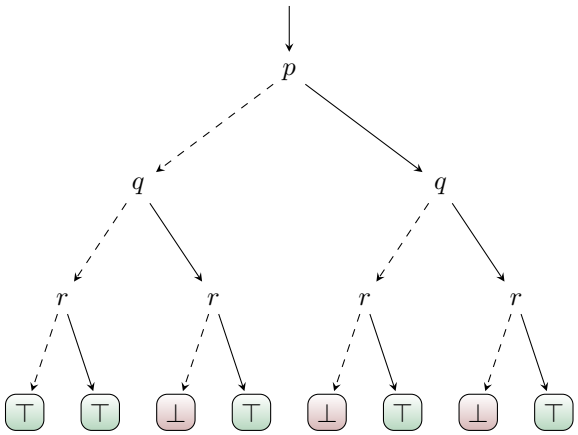
In general, we can apply two rules:

1. Merge isomorphic nodes (nodes labelled v_i whose low branches point to nodes labelled v_j and whose high branches point to nodes labelled v_k);

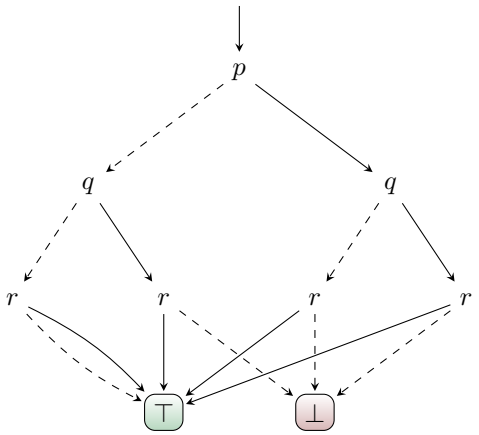
2. Remove each node whose outgoing edges point to the same child, connecting the branch that pointed from the node's parent to that node to the node's child.

We apply whichever rule can be applied, until no rules can be applied any more.

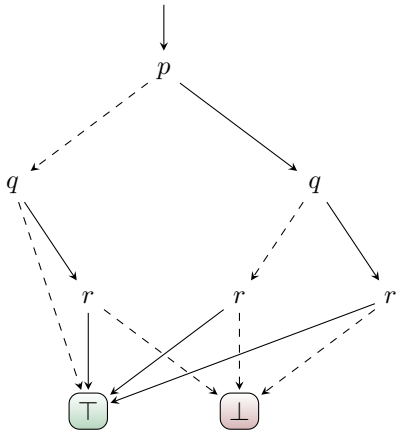
For our example, observe that in Figure 21a, we have leafs with values $\top$ and $\bot$ their outgoing branches point to the same subdiagrams (they don't have any outgoing branches, so this is automatically true), so we can apply rule 1 and merge all leafs with the same values. Doing so yields the diagram in Figure 21b. Now we see that both outgoing branches of the left r-node point to the $\top$-node. The value for r chosen on that particular path from root to leaf therefore has no influence on the end result. We can thus apply rule 2 and remove this node, obtaining the diagram in Figure 21c. Now we see that we have three r-nodes whose outgoing branches point to the same subdiagrams (the negative branch pointing to the $\bot$-leaf and the positive branch to the $\top$-leaf). These r-nodes are therefore isomorphic, and we can apply rule 1 again,



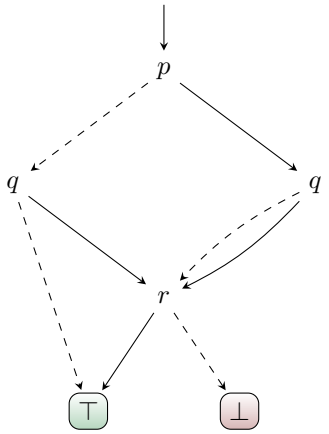
(a) A decision tree for the formula $p \vee q \rightarrow r$.



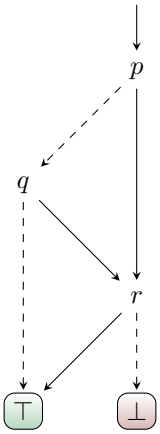
(b) Merge leafs that have the same value.



(c) Remove nodes that do not add any information.



(d) Merge isomorfic nodes.



(e) An OBDD for the formula $p \vee q \rightarrow r$.

Figure 21: Turning a decision tree (a) for the formula $p \wedge q \rightarrow r$ into an ordered binary decision diagram (e). Internal nodes represent atoms. Solid branches (*high branches*) represent the choice that the atom has truth value $\top$, dashed branches (*low branches*) represent the choice that the atom has truth value $\bot$. The child of a node pointed to by its low (high) branch is called the *low* (*high*) *child*. A path from root to leaf corresponds to an interpretation. The leafs show the truth values for $p \vee q \rightarrow r$ under the interpretations, and correspond to the values in the truth table of Table 5.

obtaining the diagram in Figure 21d. Applying rule 2 one more time yields the OBDD in Figure 21e. We have reduced a binary decision tree with fifteen nodes to an OBDD with only five.

Note that the word *order* is of importance here. For this algorithm to work, the order in which the variables are encountered in a path from root to leaf in the decision tree should be the same for all those paths. Different orders yield different OBDDs, with different levels of compression.

B.3 Sentential Decision Diagrams

Sentential decision diagrams (SDDs) [10] can be seen as a generalisation of OBDDs, in the sense that they are canonical, can be constructed using rules similar to the rule for constructing OBDDs and that any OBDD is also an SDD. While an OBDD can only branch on the truth value of *individual variables*, an SDD can branch on the truth values of *logical sentences* (formulas). Here we briefly introduce a few basic concepts regarding SDDs. For a more detailed discussion, see Darwiche's work [10].

An SDD is a data structure for Boolean formulas that is based on two concepts: that of $(\mathbf{X}, \mathbf{Y})$-*decomposition* of Boolean formulas and that of *vtrees* corresponding to those formulas.

Definition 2. *Consider a Boolean function $f(\mathbf{X}, \mathbf{Y})$ with $\mathbf{X}$ and $\mathbf{Y}$ sets of variables such that $\mathbf{X} \cap \mathbf{Y} = \varnothing$. If $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \ldots \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$, then $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ is called an $(\mathbf{X}, \mathbf{Y})$-decomposition of f. Each ordered pair (p_i, s_i) is called an* element *of the decomposition, where p_i is called a* prime *and s_i a* sub*. If $p_i \wedge p_j = \bot$ for $i \neq j$, the decomposition is called* strongly deterministic on $\mathbf{X}$*. [10, 22]*

SDDs use a more structured type of decomposition:

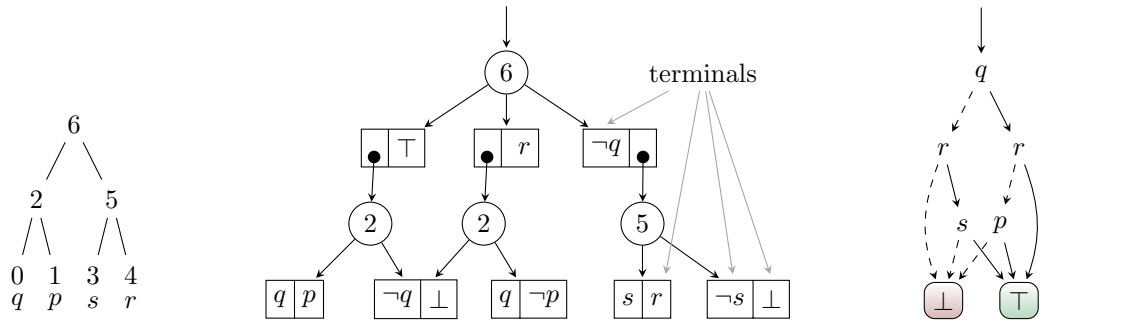
Definition 3. *Let $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ be a $(\mathbf{X}, \mathbf{Y})$-decomposition of a function f . Now α is called an* $\mathbf{X}$-partition *of f iff: [10]*

1. *each prime p_i is consistent ($p_i \not\models \bot$);*
2. *every pair of primes are mutually; exclusive*
3. *the disjunction of all primes is valid ($\bigvee_i p_i \models \top$).*

Observe that by the first condition in definition 3, $\bot$ can never be a prime, and that by the second, if $\top$ is a prime, it is the only one. It can also be shown that, if $\circ$ is a Boolean operator and $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ and $\{(q_1, r_1), \ldots, (q_m, r_m)\}$ are $\mathbf{X}$-partitions of $f(\mathbf{X}, \mathbf{Y})$ and $g(\mathbf{X}, \mathbf{Y})$, then $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \bot\}$ is an $\mathbf{X}$-partition of $g \circ f$. [10] This accounts for a lot of the properties of SDDs.

To create OBDDs, we have to define an order on the variables. SDDs use a generalisation of the concept of order: they are based on *vtrees*. A vtree is a full binary tree with variables in the leafs, each variable occurring only once. In vtrees there is a strict difference between the left child of a node and its right child. Loosely speaking: a node n_{SDD} in an SDD is said to *respect* a node n_v in a vtree, if the formula f that is represented by n_{SDD} is decomposed in such a way that:

- all the variables that occur in *primes* of the decomposition of f occur in the subtree rooted at the *left child* of n_v, and
- all the variables that occur in *subs* of the decomposition of f occur in the subtree rooted at the *right child* of n_v. [10]



(a) A vtree for variables p, q, r and s. Numbers simply represent node labels.

(b) An SDD for $(p \wedge q) \vee (q \wedge r) \vee (r \wedge s)$, respecting the vtree in (a).

(c) An OBDD for $(p \wedge q) \vee (q \wedge r) \vee (r \wedge s)$, with order $q < r < p < s$.

Figure 22: A vtree defining an order on variables p, q, r and s (a). The nodes in the SDD in (b) representing logic formula $(p \wedge q) \vee (q \wedge r) \vee (r \wedge s)$ and respecting this vtree, are labelled with the nodes in the vtree that they respect. A corresponding OBDD is shown in (c). Example from Darwiche [10].

An example of a vtree is shown in Figure 22a. Figure 22b shows an SDD for the logic formula $(p \wedge q) \vee (q \wedge r) \vee (r \wedge s)$ that respects the vtree in Figure 22a. A node n_{SDD} of the SDD is shown as a circle and is labelled with the vtree node n_v that it respects. Node n_{SDD}'s outgoing edges lead to recursively defined decompositions, for which the variables in the primes are elements of the subtree rooted at the left child of n_v and the variables in the subs are elements of the subtree rooted at the right child of n_v.

Recalling the definition of a decomposition, we see for example that node 5 in Figure 22b represents the logical formula $(s \wedge r) \vee (\neg s \wedge \bot) = s \wedge r$.

A corresponding OBDD is shown in Figure 22c. For OBDDs the size of the OBDD depends on the order that was chosen. The same holds for SDDs: a different vtree typically yields a different SDD. The *size* of an SDD is determined by its number of nodes and the number of children of those nodes. The size

of a single node is equal to its number of children. The SDD in Figure 22b has four nodes, three of which have two children and one of which has three children. Decompositions have size zero, as do terminal nodes[5]. This brings the total size of the SDD to $2 + 2 + 2 + 3 = 9$. [8]

Note that more than one formula can be turned into an SDD at the same time. In that case, a diagram is created with more than one root, where each root represents a logic formula, similar to the way that different formulas are represented in the single DAG in Figure 20. The SDD representing one particular formula may share nodes with other SDDs. The size of a single SDD is simply that of the part of the diagram that consists of the root of that SDD and all its descendants. The size entire diagram (or the *SDD manager*) is simply the sum of the sizes of all its nodes. As with OBDDs, redundancy is removed from SDDs where possible.

[5]Terminal nodes are either (negations of) single variables or constants ($\top$, $\bot$). In Figure 22b they are shown seperately, but in the SDD implementation used for this work, they are not redundantly present. [8]

C More experimental results

In this chapter we present and discuss some more of our experimental results, complementing those presented in Chapter 8.

Search time results for `fas` problems

We start with the search time results for the `fas` problem set. Figure 23 shows the search times for `fas` problems on twenty and thirty decision variables for **maxProb** optimisation problems with one and three constraints.

Looking at Figure 23, we immediately observe that our prediction (Chapter 7.3) that there is a certain window of values for ϑ for which problems are hard to solve for the DFS-based methods. In this window, little SSSP can be performed (other than for high values for ϑ, or loose constraints) and little pruning based on violated constraints (as happens for low values for ϑ, or strict constraints). We also note that search times tend to be larger for problems with three constraints than for problems with one constraint (Figures 23a and 23d), with differences up to several orders of magnitude in case of the INCREMENTAL and LAZYINCREMENTAL methods.

The results for `fas` problems in the **maxSet** optimisation setting are shown in Figure 24 do not show any surprising results. Search times generally are lower for **maxSet** problems than for **maxProb** problems, just like we saw for `mes` and `tyf` in Chapter 8.2. We continue with more results on the sizes of the SDDs that are compiled during search by the incremental algorithms.

SDD sizes from root to leaf

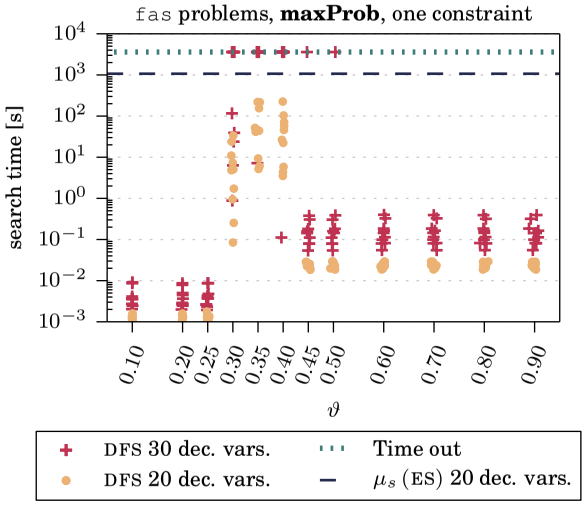
Consider the result in Figure 25. This figure shows the same largest SDD from root to leaf comparison as Figure 18 in Chapter 8.4, but for an instance of the `fas` problems on twenty decision variables that is a typical representative of these problems. We observe two mayor points in which these results differ from the ones shown in Figure 18. One is that, for `fas` problems, LAZYINCREMENTAL seems to not build smaller SDDs than INCREMENTAL on some queries (like the one in Figure 25a). The other one is that only for `tyf` problems, we observe the data point representing the solution leaf to appear under the diagonal, rather than on the diagonal. We will address these two observations in order.

When LAZYINCREMENTAL does not keep SDDs smaller

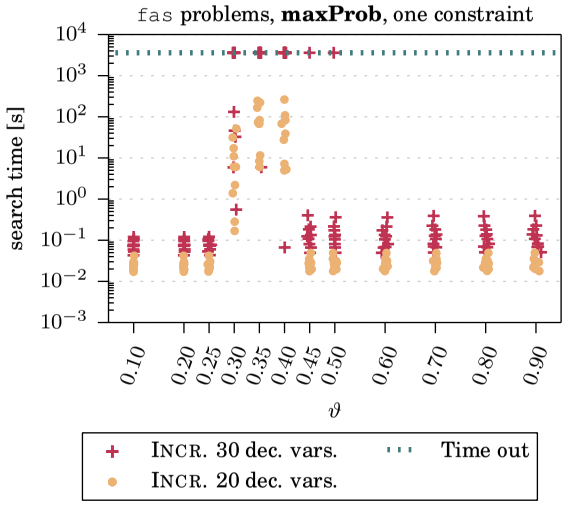
Consider the results shown in Figure 25 for the largest size of an SDD built on a path from root to leaf in the search tree (where we only consider leafs that correspond to full strategies, not nodes where the search was pruned because a partial strategy violated a constraint).

We see that a smaller fraction of the largest SDDs is smaller for LAZYINCREMENTAL than for INCREMENTAL than with the results for the `tyf` problem in Figure 18. We also see a difference in behaviour for the constraints. Specifically, we see that the largest sizes of the SDD representing the first constraint query (Figure 25a) are exactly the same for both algorithms. For the second constraint query (Figure 25b), some SDDs are smaller for the LAZYINCREMENTAL method than for the INCREMENTAL method, although the differences are but small. The largest number of SDDs that is below the diagonal is found for the optimisation query (Figure 25d), although again the differences are not very large. We also see that the SDD for the optimisation query that is built for the solution to the problem is relatively large. Finally observe that most data points in Figures 25a, 25b and 25d are located in the lower left corner: there where the SDDs are smallest.

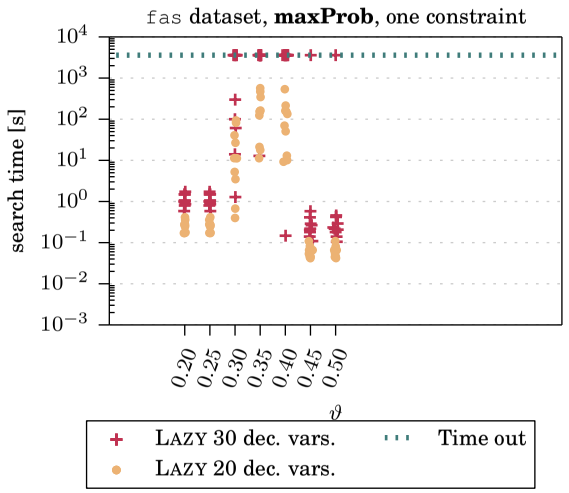
We can explain the behaviour shown in the sizes of the three constraints as follows. Recall that the LAZYINCREMENTAL algorithm first removes determinism from the first constraint, until it is either satisfied or does not contain any temporary determinism any more. The results in Figure 25 can be explained by the fact that it apparently is necessary to remove all determinism from the first constraint. Then, if it is found to be violated, it is no longer necessary to perform the same procedure for the other constraints, so they may remain smaller during the execution of the LAZYINCREMENTAL algorithm than during that of the INCREMENTAL method. The reason that all temporary determinism needs to be removed from the SDD for first constraint is as follows. The first constraint is $P(smokes(x_1)) \leq 0.4$. Node x_1 has two neighbours in the graph underlying the problem of Figure 25: x_0 and x_6. If the decision variables d_{01} and d_{61} representing the incoming edges of x_1 are chosen to be $\top$, the probability that the



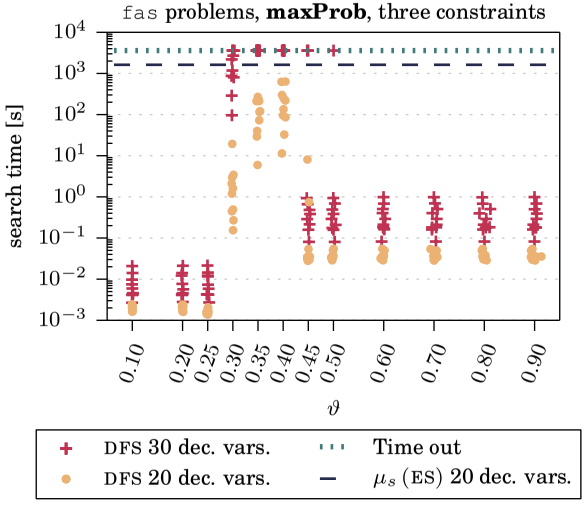
(a) ExhaustiveSearch and dfs, `fas` problem set, one constraint, μ_s (ES) $= 1.1 \pm 0.3 \cdot 10^3$ s.



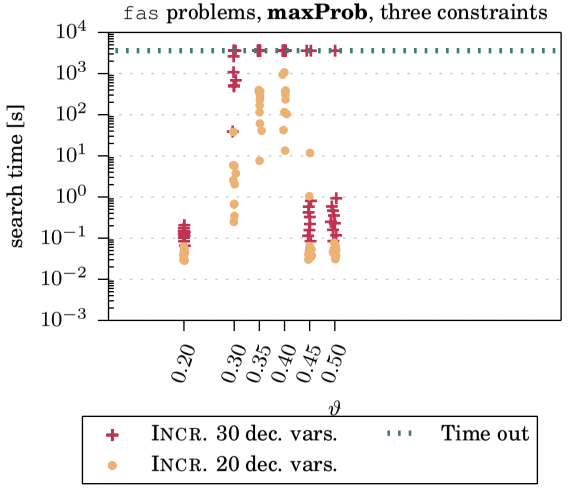
(b) Incremental, `fas` problem set, one constraint.



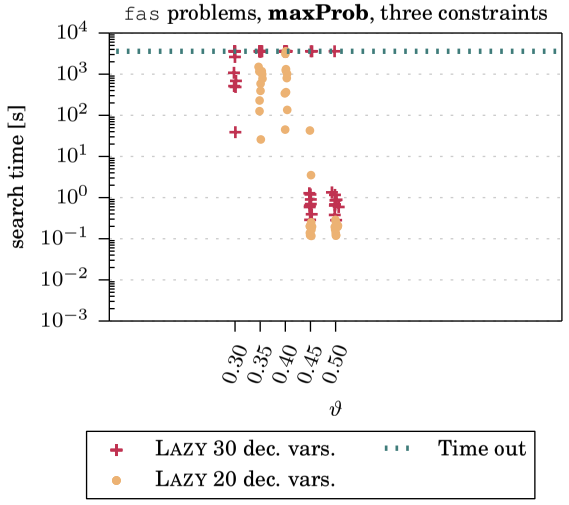
(c) LazyIncremental, `fas` problem set, one constraint.



(d) ExhaustiveSearch and dfs, μ_s (ES) $= 1.6 \pm 0.7 \cdot 10^3$ s.

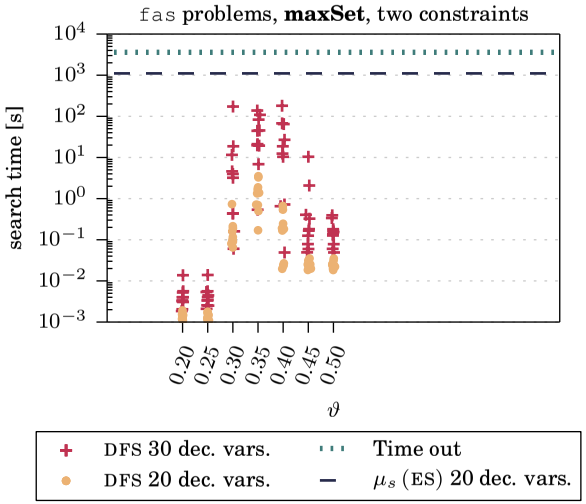


(e) Incremental, `fas` problem set, three constraints.

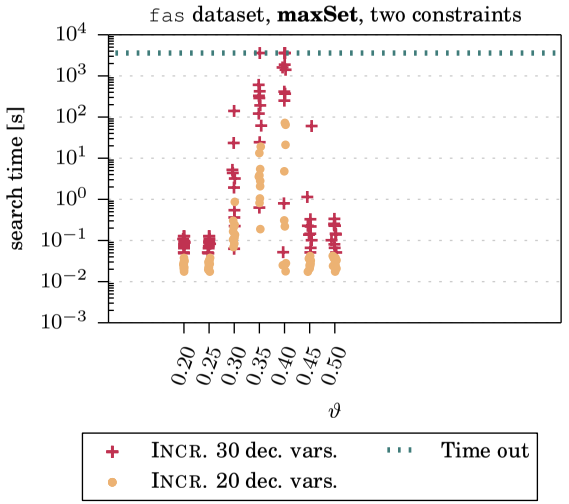


(f) LazyIncremental, `fas` problem set, three constraints.

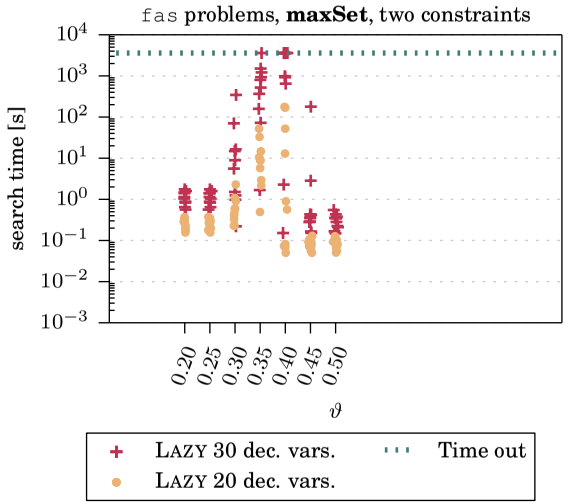
Figure 23: Search times for ExhaustiveSearch and dfs (left), Incremental (middle) and LazyIncremental (right) for `fas` problem set in **maxProb** setting with one constraint (top) and with three constraints (bottom). Jitter added in horizontal direction to separate datapoints.



(a) ExhaustiveSearch and dfs, `fas` problem set, two constraints, μ_s (ES) $= 0.5 \pm 0.7 \cdot 10^3$ s.

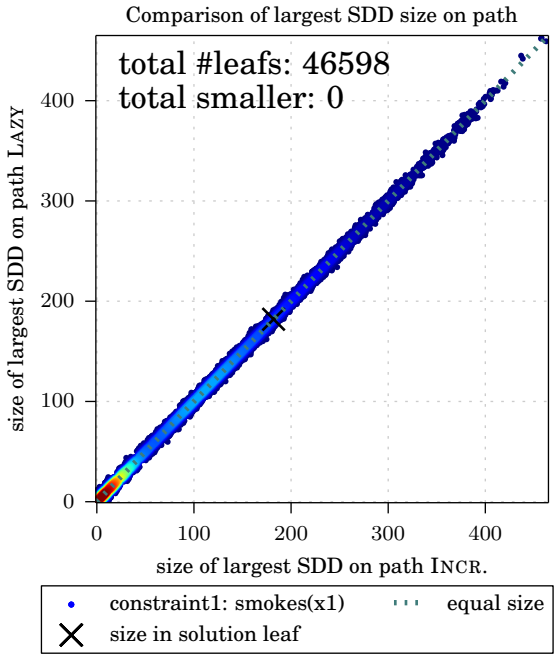


(b) Incremental, `fas` problem set, two constraints.

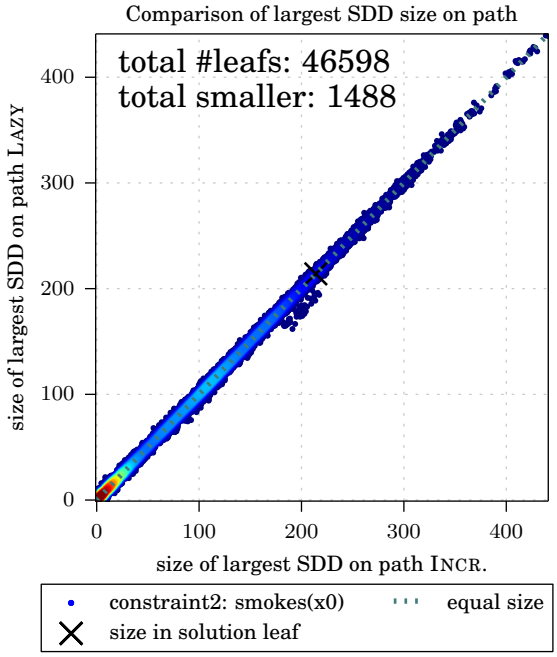


(c) LazyIncremental, `fas` problem set, two constraints.

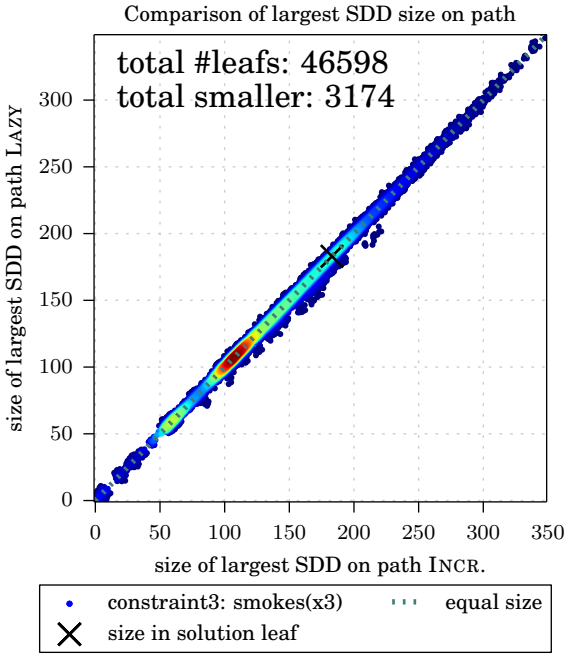
Figure 24: Search times for ExhaustiveSearch and dfs (left), Incremental (middle) and LazyIncremental (right) for `fas` problem set, in the **maxSet** setting with two constraints. Jitter added in horizontal direction to separate datapoints.



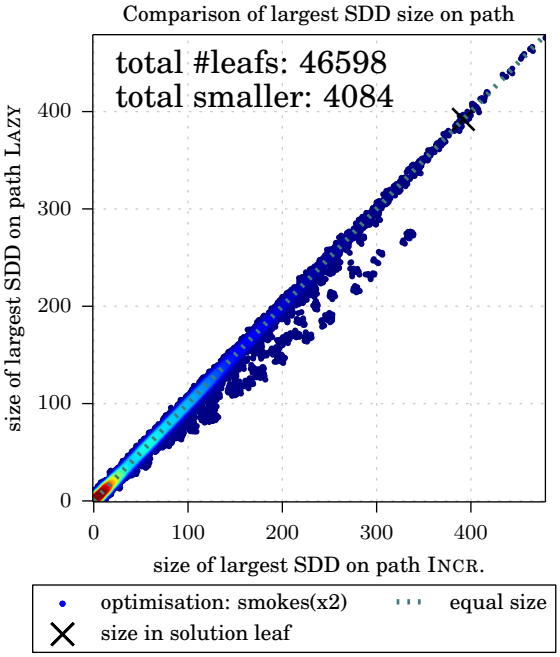
(a) Sizes of SDD representing the first constraint query.



(b) Sizes of SDD representing the second constraint query.



(c) Sizes of SDD representing the third constraint query.



(d) Sizes of SDD representing the optimisation query.

Figure 25: Comparison of largest SDD encountered on each path from root to leaf, by the INCREMENTAL method (horizontal axis) and the LAZYINCREMENTAL method (vertical axis). Results are shown for problem instance 7 of the `fas` problems on twenty decision variables, for the constraint queries (a)—(c) and the optimisation query (d). Note that (c) has a slightly different range of SDD sizes than the other figures.

person represented by x_1 smokes, is given by:

$$P\left(sm(x_1)\right) = 1 - ((1 - P\left(str(x_1)\right)) \cdot (1 - P\left(sm(x_0)\right) P\left(infl(x_0, x_1)\right)) \cdot (1 - P\left(sm(x_6)\right) P\left(infl(x_6, x_1)\right)))\,,$$

with sm, str and $infl$ abbreviations for $smokes$, $stressed$ and $influence$. If all relevant probabilistic variables are temporarily deterministic, the probability that x_1 smokes is:

$$P\left(sm(x_1)\right) = 1 - ((1-1)\cdot(1-1\cdot 1)\cdot(1-1\cdot 1)) = 1.$$

Clearly, the constraint is violated. The LazyIncremental proceeds to remove the temporary determinism from the SDD that represents query $smokes(x_1)$, starting with the probabilistic variables that have the smallest probabilities. In this case, these are the variables that correspond to the $influences(x, y)$ predicates. If all these are set to their true probabilistic values (0.2), the probability that x_1 smokes is as follows:

$$P\left(sm(x_1)\right) = 1 - ((1-1)\cdot(1-1\cdot 0.2)\cdot(1-1\cdot 0.2)) = 1.$$

Now, one by one, the temporarily deterministic variables representing the probability that people are stressed are set to their probabilistic values. Suppose this process starts with the probability that x_1 themself is stressed. The resulting probability of x_1 being a smoker is:

$$P\left(sm(x_1)\right) = 1 - ((1-0.3)\cdot(1-1\cdot 0.2)\cdot(1-1\cdot 0.2)) = 1 - \left(0.7\cdot 0.8^2\right) = 0.552.$$

Still the constraint is violated, so LazyIncremental continues removing determinism. Suppose x_0 is stressed but has no people that consider them a friend, this brings the probability of x_1 being a smoker to:

$$P\left(sm(x_1)\right) = 1 - ((1-0.3)\cdot(1-0.3\cdot 0.2)\cdot(1-1\cdot 0.2)) = 1 - (0.7\cdot 0.94\cdot 0.8) = 0.4736,$$

still violating the constraint. Only if the probability that also x_6 is stressed, is no longer deterministically equal to 1, the constraint may be satisfied (under the optimistic assumption that x_6 is also friendless and therefore has a probability of 0.3 that they smoke):

$$P\left(sm(x_1)\right) = 1 - ((1-0.3)\cdot(1-0.3\cdot 0.2)\cdot(1-0.3\cdot 0.2)) = 1 - \left(0.7\cdot 0.94^2\right) = 0.38148.$$

We conclude that if both incoming edges for the constraint are chosen to be $\top$, LazyIncremental has to reset at least $n_{f,C}+3$ temporarily deterministic probabilistic variables to their true probabilistic values in order to satisfy the constraint (with $n_{f,C}$ the number of friendships that exist for partial assignment C). Observe that even if x_1 has no friends in partial assignment C, the SDD representing query $smokes(x_1)$ consists of only a terminal node that represents $stressed(x_1)$, which has the same size if it is temporarily deterministic, as it then consists of only terminal node $\top$.

Largest SDD on path from root to solution leaf

Note that in all figures in Figure 25, the solution leaf is located on the diagonal. This is what we expect for the optimisation criterion, as all the determinism needs to be removed from the SDD that represents the optimisation query in order to know for sure that a found value is better than previously found values. The fact that the same holds for the SDDs representing the constraint queries, is explained by the characteristics of the `fas` problems. Since the networks representing the graphs are strongly connected, *all* probabilistic variables are relevant to *each* possible query. Therefore, when removing determinism from the SDD that represents the optimisation query, the same determinism is also removed from the SDDs representing the constraint queries.

So why did we observe different behaviour in the `tyf` results of Figure 18? The answer lies in the characteristics of the `tyf` problems. As described in Chapter 5, we select target nodes for queries such that there exists at least one path of length three from source to target. However, the network remain rather small, because otherwise the problems get too large for our experiments. It thus may happen that a target node t is chosen that also happens to be a neighbour of the source node s.

Consider a query $P\left(path(s,t) \mid T, \sigma\right)$ for a `tyf` problem. Somewhere on a path from root to leaf in the search tree, a proof for $path(s,t)$ is found, while d_{st}, the decision variable for the edge from s to t is still unassigned, and therefore assumed to be $\bot$. Suppose also that the SDD built by Incremental is larger than the SDD built at the same node in the search tree by LazyIncremental, because the latter still contains temporarily deterministic probabilistic variables because no constraints are violated.

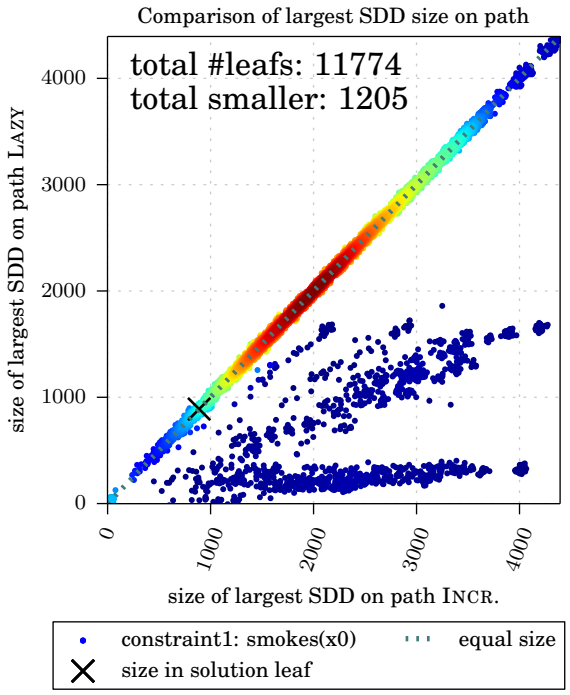
Now suppose that the algorithms move on in the search tree, branching on $\top$ for d_{st}. Now $path(s,t) = \top$, which means that the SDD representing the formula is simply that of $\top$, and therefore contains no variables, and thus no temporary determinism in the case of LazyIncremental. When the search process now ends up in a leaf node that corresponds to a total choice, the size of the largest SDD built

by the INCREMENTAL algorithm to get to this leaf is larger than the largest SDD that was built by LAZYINCREMENTAL, hence the corresponding data point is located under the diagonal.

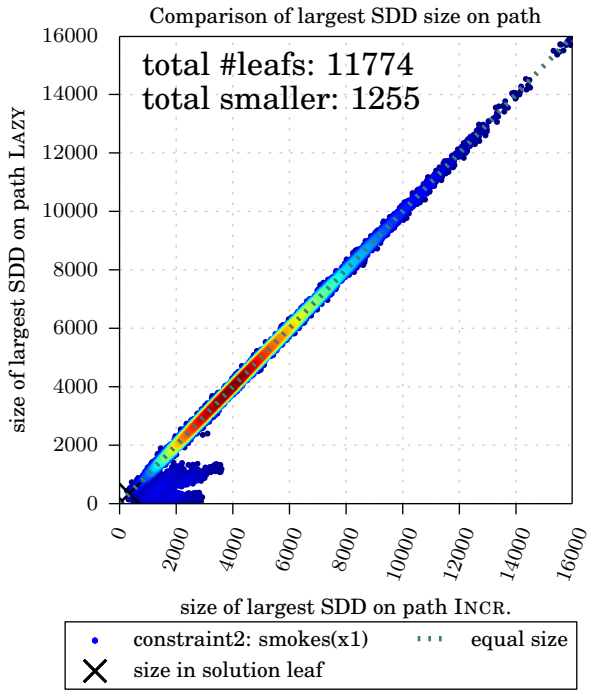
We conclude that the reason that this is observed for `tyf` problems and not for `fas` and `mes` problems, is that in the `tyf` problems it is possible to be certain that there is a path from source to target, while for the `fas` and `mes` problems, this is not possible.

Other observations on SDD sizes

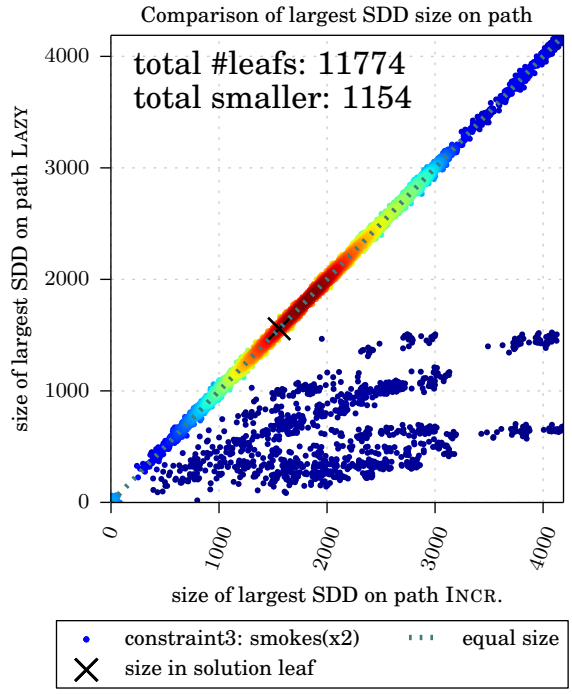
Figure 26 shows results for a `fas` problem on thirty decision variables, in the **maxSet** optimisation setting with three constraints. What we observe in these figures, is some structure in the plot: we see lines in the data points below the diagonal. This could indicate that entire subproofs are never compiled by LAZYINCREMENTAL for certain strategies. Note that the scale of Figure 26b is rather different than the scales of the other two figures. It is remarkable that it is for this particular constraint, that the largest SDDs that was compiled on the path from root to solution leaf, is very small indeed.



(a) Sizes of SDD representing the first constraint query.



(b) Sizes of SDD representing the second constraint query.



(c) Sizes of SDD representing the third constraint query.

Figure 26: Comparison of largest SDD encountered on each path from root to leaf, by the INCREMENTAL method (horizontal axis) and the LAZYINCREMENTAL method (vertical axis). Results are shown for problem instance 1 of the `fas` problems on thirty decision variables, in the **maxSet** optimisation setting, so figures show SDDs for the three constraints only. Note that (b) has a very different range of SDD sizes than the other figures.