MONASH University

Modelling and Solving Techniques for Stochastic Combinatorial Optimisation Problems

by

David Hemmi

MSc Robotics

A thesis submitted for the degree of Doctor of Philosophy at

Monash University in 2019

Caulfield School of Information Technology

Supervisors:

Dr. Guido Tack

Prof. Dr. Mark Wallace

Contents

List of Tables

List of Figures

Modelling and Solving Techniques for Stochastic Combinatorial Optimisation Problems

David Hemmi
david.hemmi@monash.edu
Monash University, 2019

Supervisor: Dr. Guido Tack
[redacted]
Associate Supervisor: Prof. Dr. Mark Wallace
[redacted]

Abstract

Combinatorial optimisation is concerned with solving mathematical problems that are characterised by discrete choices, a set of constraints and an objective function, which measures the quality of a solution. Solving these problems is critical in a wide range of areas, such as telecommunications, circuit design, supply chain management and transportation. However, combinatorial optimisation problems are often subject to uncertainties that have to be taken into account to produce realistic solutions.

Stochastic Programming is concerned with solving optimisation problems under uncertainty. First, decisions are implemented before observing the random variables, and thereafter recourse actions are taken in response to the now observed random variables. Solving stochastic programs, especially of combinatorial nature, is notoriously difficult. However, numerous systems emerged over the years that enable optimisation practitioners to A) model decision problems under uncertainty in a high-level language and B) use back-end solvers to find high quality policies.

This thesis is concerned with improving systems that facilitate the modelling and solving of combinatorial stochastic programs and is composed of four main contributions. First, we show how symmetry and dominance breaking constraints are to be used correctly when modelling stochastic problems and study other techniques to improve stochastic optimisation models. Secondly, we improve the performance of an algorithm introduced by Ahmed (2013) that is based on the scenario decomposition and developed to solve two-stage stochastic problems with binary first-stage variables. Thirdly, we show how the same algorithm can be generalised for multi-stage problems. And lastly, we present a method that combines optimisation and simulation to find strong policies for problems that are described using a large set of scenarios.

In conclusion, we show how modelling frameworks for stochastic programming can be improved by incorporating modelling techniques from Constraint Programming and by developing algorithms that utilise off-the-shelf solvers.

Modelling and Solving Techniques for Stochastic Combinatorial Optimisation Problems

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

David Hemmi
February 17, 2019

Acknowledgments

I owe a great debt of gratitude to the supportive environment at Monash University, my family and friends whose backing was instrumental for completing this work. I completed a PhD, but lasting will be the memories and friendships created during the time of my candidature.

First and foremost, I would like to thank my supervisors, Guido Tack, Mark Wallace, and Andrea Rendl, for their overwhelming support. Without Andrea, I would have not started my studies at Monash University. She has been a great inspiration ever since I met her as a trainee at the Austrian Institute of Technology (AIT) in Vienna. I thank Guido and Mark for their advice, encouragement, feedback and their generosity with time throughout the whole PhD process.

Lots of learning and personal growth over the past years is attributed to extracurricular activities and a big thank goes to Danushi Peiris for working with me on such exciting projects.

I'm eternally grateful to my flatmates; this wasn't just a house, it was a home! Experiencing such incredible support was and still is overwhelming, thank you David, Richard, Leslie, Helen and the whole family Graber for all you have done for me.

My sincere thanks also go to the people of Australia, Data61/NICTA and Monash University for allowing me to be part of the vibrant Australian research community. I'd also like to thank my colleagues and friends at Monash University for stimulating discussions and the great vibe in our office. Especially, I would like to mention the support of Kevin Leo. Furthermore, I enjoyed many valuable and inspiring conversations with academics that greatly helped shaping my thesis, I'd like to thank: Peter Stuckey, Tias Guns and Ahmed Shabbir.

Last but not the least, I wish to thank my parents, Marianne and Hansjörg and my sister Fiona for their continued support.

David Hemmi

Monash University
February 2019

Chapter 1

Introduction

Complex decision making is ubiquitous in our modern society. Establishing a strategic policy for long term investments, such as designing a future-proof electricity network or transport system, is very challenging, since analysing large amounts of data is necessary to make an informed statement. Similarly, operational decisions, such as packing cargo onto delivery trucks to minimise the cost of distribution or scheduling operations, are decision problems that require to be solved very quickly as they have to be implemented without delay.

There is a rich history of using advanced analytical methods to help make better decisions. These techniques might assist a decision maker in understanding the impact of an action better, e.g. by using simulation or visualisation, or aim to improve the quality of the decision-making process by computationally analysing data and intelligently proposing a course of action. In particular, the field of mathematical optimisation is concerned with finding a policy that satisfies a set of constraints while maximising or minimising a value function. A well-known example that is solved using mathematical optimisation is the travelling salesman problem where a list of cities is given to a sales person who decides in what order to visit all of them before returning to the starting point while minimising the total distance travelled.

Various research directions that aim to improve mathematical programming have emerged over the years, for example, linear programming, mixed-integer programming, and constraint programming to name just few. The structural properties of a decision problem determine which technology is most promising for finding good policies quickly.

In practice, decision makers often face choices in environments that are governed by uncertainty. For example, it is difficult to forecast the expected electricity consumption in twenty years' time, or even what the weather will be like next week, which greatly determines the energy production level of a solar or wind power plant. These uncertainties must be taken into account when investing in the electricity network or managing energy production. Similarly, parameters such as customer demand, travel times or the price of resources are often unknown at the time a decision is made.

This thesis contributes to decision making under uncertainty by combining ideas from constraint programming and stochastic programming. The field of stochastic programming is concerned with solving decision problems that are subject to uncertainty.

Constraint programming is looking at modelling and solving decision problems with discrete choices, also called combinatorial optimisation problems.

In this thesis, we focus on modelling and solving combinatorial optimisation problems under uncertainty. We contribute by proposing modelling techniques that improve stochastic problem formulations and by developing algorithms that find high quality policies for stochastic problems quickly. The subsequent sections will look more closely into stochastic programming, introduce key terms, and define the scope and contribution of this thesis in more detail.

1.1 Modelling and Solving Stochastic Programs

A key factor that enabled the success of classical (deterministic) mathematical programming is the separation of modelling and solving a decision problem. To formalise mathematical problems, a whole range of high-level modelling languages are available that allow a decision maker to write models at a high level of abstraction. Often, these modelling systems are solver agnostic, which means that a problem can be formalised without being tailored for a specific solver. To solve the mathematical model, a back-end solver, which is essentially an algorithm that is able to interpret and solve the model is utilised. Since the model is solver agnostic, the decision maker can experiment with a variety of solvers that have different strengths and weaknesses to find the technology that works best to solve a specific optimisation problem. This separation enables decision makers to formalise and solve a mathematical problem without having to implement algorithms themselves.

Decision problems under uncertainty are different to deterministic (no uncertainty) decision problems as they contain random variables that describe the uncertainty. It is possible to model and solve stochastic programs using standard modelling systems as described above. However, this is not advisable (Watson et al.; 2012), as it requires understanding of how to formalise a mathematical program that contains random variables. Furthermore, and perhaps more importantly, standard back-end solvers are not well equipped to solve stochastic programs. For this reason, multiple modelling frameworks for decision making under uncertainty have been developed over the years.

One way of modelling a stochastic program is by separating the problem into three key components as illustrated in Fig. 1.1; a decision model, an uncertainty model and use case specific data. As the name implies, the decision model contains a parametric description of the decisions, constraints and objective function that define the optimisation problem. The uncertainty model characterises the random variables, using scenarios, and their impact on the decision model. A scenario is a concrete instantiation of the random variables and a set of scenarios is used to describe our expectation of the future. The decision model is impacted by the uncertainty model as the latter assigns decisions to time stages. For example, in a two-stage stochastic program, decisions are taken before observing the random variables, as well as afterwards. We call them first- and second-stage decisions and the same concept can be generalised for multiple (more than two) decision stages. And finally, the data file describes a specific use case of the model and matches the parameters

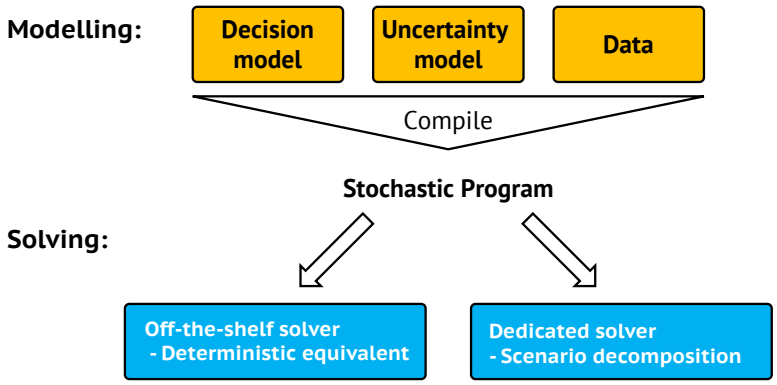


Figure 1.1: Model and Solve Stochastic Problems

in the decision and uncertainty model. The three components are then compiled into an instance of a stochastic program.

To provide a better intuition for the three key components required to model a stochastic program, we will use an example. Consider a news vendor who has to decide early in the morning how many newspapers to purchase from the wholesaler, to sell them afterwards, on that very same day to prospective customers. If the demand is less than the number of copies she bought, she can return the left-overs at a low price. If she runs out of news papers during the day, she pays a penalty. Her goal is to maximise the profit of selling newspapers to customers. The **decision model** contains precisely what has just been described; a decision, how many copies to purchase, a set of rules, what happens if the number of copies she purchased does not match the demand, and her objective, maximising profit. In her case however, the customer demand is uncertain and the **uncertainty model** contains a description of the random variables, e.g. the probability distribution that describes how many copies she is likely to sell. The uncertainty model also assigns the purchase decision to the first stage and describes that she wants to maximise her expected profit. *The expected profit denotes the average profit she makes when purchasing the same number of news papers over many similar days.* And lastly, the data file is used to create a specific instance, for example, there might be multiple news vendors who apply our model to their specific use case.

An instance of a stochastic program is solved in one of two ways. A straightforward method to solve the stochastic problem is by formalising the **deterministic equivalent** (DE), which is solved using an off-the-shelf solver. The DE is a single model that contains all decisions and scenarios. However, while this approach is convenient, as it utilises standard solver technology, its performance is often poor (Birge and Louveaux; 2011). To achieve better performance, dedicated techniques to solve stochastic programs have been developed (Schultz; 2003). These specialised algorithms are able to exploit the model structure of a stochastic program, by decomposing the problem into smaller subproblems that can be solved quickly.

1.2 Scope

This thesis is concerned with modelling and solving combinatorial optimisation problems under uncertainty. The term combinatorial optimisation refers to decision problems with discrete choices, for example assigning a lecturer to a classroom or deciding which delivery truck will visit a customer. Discrete choice models with random variables are extremely challenging to solve, as the complexity of a mathematical model increases greatly when scenarios are used to describe the random variables.

The techniques used to solve combinatorial problems have various origins. On the one hand, methods such as linear programming (LP) and its extension mixed integer programming (MIP) are associated with applied mathematics and operations research. On the other hand, approaches such as constraint programming and constraint logic programming originate in computer science. Until now, most of the work in stochastic programming has been closely linked to LP and MIP techniques.

In this thesis we will improve on some of the existing MIP techniques, translate concepts from constraint programming to stochastic programming and develop new methods that are a combination of both worlds. We focus on improving existing frameworks for modelling and solving stochastic programs. We will not propose a new modelling paradigm or framework, but rather acknowledge that there are multiple systems available that already cater for uncertainty. We address the modelling of stochastic programs by proposing techniques to strengthen decision models formalised in one of these existing systems; a model is stronger if it can be solved faster. However, we will not address how to formalise uncertainty or generate scenarios, as this is a research topic in itself. Furthermore, we address the solving of stochastic programs by developing algorithms to solve combinatorial stochastic programs and evaluate their performance using empirical experiments.

1.3 Research Questions and Objectives

The **research objective** of this thesis is to improve the performance of systems that support decision making under uncertainty. We aim to develop methods and algorithms that enhance the usability of modelling frameworks developed to solve stochastic programs. All our research efforts are made with Stochastic MiniZinc (Rendl et al.; 2014) in mind, a modelling framework for stochastic programs; however, our results are equally applicable to be implemented in comparable systems. The research questions (RQ) that govern this thesis are the following.

1.3.1 RQ1: What role can modelling techniques that have shown to improve deterministic optimisation problems play when modelling stochastic programs?

In constraint programming, multiple techniques to strengthen optimisation models have been proposed. For example, combinatorial problems often exhibit symmetries and dominance relations. Two solutions are symmetric if they are invariant to transformations,

in other words the quality of two solutions is equal. A dominance relation describes a situation where one solution is guaranteed to be better than another. Breaking symmetries and dominance relations is standard practice in traditional (deterministic) constraint programming. In this thesis we will explore the following hypothesis:

Hypothesis 1:

Constraint programming modelling techniques that are evidently beneficial to solve deterministic combinatorial problems can be adopted to strengthen stochastic programs.

1.3.2 RQ2: How can scenario based combinatorial stochastic programs, formalised in a high-level modelling language, be solved effectively using decomposition methods and off-the-shelf solvers?

As show in Fig. 1.1, there are two ways of solving a stochastic program. The most straightforward way, is to compile the stochastic program into a deterministic equivalent and solve it with an off-the-shelf solver. However, while the DE method works well to solve easy problems, often times it does not scale to more difficult ones. For this reason, many researchers have proposed algorithms that exploit the specific problem structure of stochastic programs. A common approach that is based on model structure analysis is called **decomposition**, where the original problem is broken up into subproblems that can be solved with ease. A stochastic problem can for example be separated by scenarios, where each scenario is considered as a subproblem. Many algorithms to solve stochastic problems are developed on the basis of the scenario decomposition (Schultz; 2003).

In this thesis, we are looking to exploit the scenario decomposition to solve stochastic problems effectively. We aim to incorporate learning techniques from constraint programming, generalise a two-stage algorithm to solve multi-stage problems and propose techniques to improve existing algorithms, which leads us to the following hypothesis.

Hypothesis 2:

Combining the scenario decomposition with off-the-shelf solvers allows us to develop algorithms that effectively solve combinatorial stochastic optimisation problems to optimality.

1.3.3 RQ3: How can simulation be used to improve the quality of a policy?

We are interested in solving stochastic programs, where in many cases, the scenario-based uncertainty model is only an approximation of the actual stochastic process. For example, a continuous distribution is sampled to obtain scenarios. On the one hand, the more scenarios are used to describe the uncertainty, the better the approximation (Shapiro and Philpott; 2007). On the other hand, increasing the number of scenarios also increases the time it takes to solve a stochastic program (Kleywegt et al.; 2002). In this thesis we are interested in combining optimisation techniques to solve scenario based stochastic

problems with simulation, to achieve a trade-off between the time to solve a stochastic problem and the quality of the uncertainty approximation.

Hypothesis 3:

Combining optimisation and simulation techniques allows us to find policies that are better with respect to the actual stochastic process, rather than the scenario-based uncertainty model, which is only an approximation thereof.

1.4 Structure and Contributions

This thesis is structured to first provide the relevant background knowledge before addressing the research questions in sequence.

1.4.1 Chapter 2: Background

The background chapter introduces the fundamental concepts of deterministic optimisation, such as constraint programming, mixed-integer programming and relevant modelling techniques. It proceeds by addressing the basics of stochastic programming and relates it back to deterministic optimisation. Thereafter the background introduces the formal notation that is used throughout the thesis for developing algorithms. The chapter concludes with the presentation of an algorithm to solve two-stage stochastic problems introduced by Ahmed (2013) that is relevant for all subsequent chapters.

1.4.2 Chapter 3: Modelling

The modelling chapter aims to improve the performance of solving stochastic programs using modelling techniques; it discusses the role of adding extra constraints to stochastic programming formulations that are not strictly required for the basic problem description. First, we discuss the role of symmetry and dominance relations in constraint programming. It is well established that using symmetry and dominance breaking constraints yields stronger models. However, these constraints are not an integral part of the model description, rather they aim to support the solver to search more effectively. Secondly, when decomposing a stochastic program, information that was implicit before splitting up the model is no longer available in the subproblems. We introduce a modelling technique, based on redundant constraint, that explicitly introduces this information back into the subproblems.

Contribution

We show that explicitly marking constraints such as dominance and symmetry breaking constraints, is a necessity in stochastic programming modelling frameworks. Failing to identify these extra constraints explicitly leads to incorrect model definitions. We also show how to compile the extra constraints once they are marked as such.

Secondly, we show how problem specific information is lost when decomposing a stochastic program into scenarios. Thereafter, we propose a solution and empirically demonstrate its effectiveness.

1.4.3 Chapter 4: Evaluate and Cut with Diving

This chapter is concerned with solving two-stage combinatorial stochastic programs. The background chapter introduces an algorithm that we call Evaluate and Cut published by Ahmed (2013). This algorithm is based on the scenario decomposition and performs very strongly in finding the optimal solution to stochastic programs quickly. However, the algorithm exhibits a weakness with regard to proving optimality.

Contribution

We present a method called *diving* to decrease the time it takes to prove optimality and ultimately improve the performance of Evaluate and Cut. Furthermore, we introduce *vertical learning*, an application of inter-instance learning, a constraint programming technique introduced by Chu and Stuckey (2012). Finally, we evaluate our contribution using empirical experiments. This chapter has been published as:

> David Hemmi, Guido Tack, and Mark Wallace. "Scenario-based learning for stochastic combinatorial optimisation". *Integration of AI and OR Techniques in Constraint Programming*, CPAIOR *2017, Proceedings, pp. 277-292.*

1.4.4 Chapter 5: Recursive Evaluate and Cut

This chapter is concerned with solving multi-stage combinatorial stochastic programs. Often, uncertainty is revealed over time and modelling a stochastic program with multiple decision stages is natural. Solving multi-stage problems is challenging and adapting algorithms that have been developed to solve two-stage problems is not straightforward.

Contribution

We present an algorithm called *Recursive Evaluate and Cut* that is a generalisation of the two-stage algorithm introduced by Ahmed (2013). First, we present the naïve recursive of the two-stage algorithm. Thereafter, we present an improved recursive algorithm that solves combinatorial multi-stage programs effectively. Finally, we solve a multi-stage facility location problem to provide empirical evidence for the performance of Recursive Evaluate and Cut. This chapter has been published as:

> David Hemmi, Guido Tack, and Mark Wallace. "Decomposition-Based Solving Approaches for Stochastic Constraint Optimisation". *Thirty-Second Conference on Artificial Intelligence*, AAAI *2018, Proceedings, pp. 1322-1329.*

1.4.5 Chapter 6: Simulation Based Evaluate and Cut

This chapter presents a method that combines Evaluate and Cut (Ahmed; 2013), an optimisation technique, with simulation. Often when using scenario based stochastic programming, a trade-off is made between A) the accuracy of the uncertainty model that approximates the real stochastic process and B) the time required to solve the resulting optimisation problem. More scenarios improve the quality of the approximation, but also increase the time it takes to solve a stochastic program.

Contribution

We present an algorithm called *Simulation Based Evaluate and Cut* that combines standard Evaluate and Cut with simulation techniques to get the best out of both worlds; the optimisation algorithm finds high quality policies, and simulation is used to evaluate the quality of the resulting policies on a model that approximates the stochastic process better compared to the optimisation model.

1.4.6 Chapter 7: Conclusions

The final chapter summarises the work that was completed as part of this thesis, presents conclusions and addresses future work.

Chapter 2

Background

2.1 Introduction

This thesis contributes to the modelling and solving of combinatorial optimisation problems that are subject to uncertainty. The overarching aim is to solve stochastic programs faster. This is achieved by improving modelling and solving techniques that are suitable to tackle stochastic combinatorial optimisation problems.

This chapter introduces the fundamental concepts required to understand the subsequent contributions. We start with a non-formal introduction to combinatorial optimisation and show how to model and solve these types of problems. Thereafter, we proceed with decision making under uncertainty, specifically, we introduce stochastic programming.

The first part of the chapter, namely sections 2.2 to 2.6.3, aims to introduce the relevant background without using mathematical formulations. The structure was chosen with purpose, with the aim to be accessible to the reader who wishes to get an overview of the research project without studying the formalities of the thesis in great mathematical detail. The chapter concludes with section 2.7 that introduces the mathematical notation before formally introducing the most relevant background algorithms. Out of necessity, the chapter repeats certain topics; a topic may initially be addressed informally in the first part before looking at the same topic once again using mathematical notation.

2.2 Combinatorial Optimisation

Combinatorial optimisation is a field at the intersection of applied mathematics and theoretical computer science and is concerned with finding an optimal selection from a finite set of objects that maximises or minimises an objective function. Combinatorial problems are found in a wide range of areas such as circuit design, transportation, network management and configuration, logistics, and production scheduling (Barták; 1999). In this thesis, we will investigate the impact of uncertainty when modelling and solving combinatorial optimisation problems. For example, finding a production schedule when processing times are uncertain is quite a different problem compared to a production scheduling problem with known processing times.

There are many ways to solve combinatorial optimisation problems. For example, one could implement a problem specific algorithm. The advantage of problem specific algorithms is that they are able to exploit what is known about the problem and can therefore be very efficient. However, implementing a special algorithm to solve a problem can often not be justified, as the time to develop, test and deploy a new method is high. To decrease the effort of solving optimisation problems, general purpose solvers have been developed. A solver is designed to read a mathematical formulation and to solve it. Mature solvers have typically many decades of research and development invested in them.

Using general-purpose solvers breaks up the task of solving combinatorial problems into two parts. Firstly, an active field of research is concerned with the question of how to increase the efficiency of general-purpose solvers. Secondly, questions arise on how to model an optimisation problem, as a well formalised model reduces the time it takes the solver to find high quality solutions.

While generic solvers are able to solve many kinds of combinatorial problems, they often work best for a specific problem class and not so well for others, a classic example of the no-free-lunch theorem (Wolpert and Macready; 1997). It is often difficult to predict what kind of algorithm or solver works best to solve a specific problem (Kotthoff; 2016). Therefore, having a system that enables researchers and practitioners to use several different solvers without rewriting their model is of great value.

The rest of this section introduces the most relevant solving techniques for this thesis, without aiming to be comprehensive.

2.2.1 Mixed-Integer Programming

Mixed-integer (linear) programming (MIP) is concerned with solving combinatorial optimisation problems that are formalised using a mix of continuous and discrete variables constrained by linear inequalities. The quality of a solution is measured using a linear objective function. MIP is an extension of linear programming (LP), a line of research that is said to have started with George Dantzig in 1947 (Bixby; 2012). LP is concerned with solving problems that are formalised using only continuous variables and linear constraints.

Equation 2.1 is a simple example of a MIP. The problem is composed of two integer variables x and y, a set of four linear constraints and an objective function that asks to maximise the value of y.

$$\begin{aligned} \text{maximise: } & y \\ -x + y \leq & 1 \\ 3x + 2y \leq & 12 \\ 2x + 3y \leq & 12 \\ x, y \leq & 0 \\ x, y \in & \mathbb{Z} \end{aligned} \tag{2.1}$$

The main steps to solve the problem introduced above are displayed in Fig. 2.1. First, by removing the integrality condition on the variables x and y we obtain the LP relaxation of

the MIP problem. The LP relaxation is considered easy to solve; it can be solved using the well-known simplex algorithm or the interior point method. Note, the optimal solution to an LP can always be found at an intersection of two or more constraints, depending on the dimensions of the problem.

In our example, the optimal solution to the LP relaxation assigns fractional values for both variables x and y (1). To obtain an integer solution, a branching scheme that splits the original problem into two subproblems is used. By introducing the constraints $x \leq 1$ and $x \geq 2$ (2), two subproblems are generated that rule out a current non-integer value without eliminating any integer solutions. After introducing the additional constraints, the LP relaxation for the subproblem with the additional constraint $x \geq 2$ is solved. The new solution is integral in x, however remains fractional in y. This is why we branch on y and solve the LP relaxation again to obtain a feasible solution; $x = 2$ and $y = 2$ (3). Lastly, after finding an incumbent solu@tion, the feasible solution with the highest value seen so far, we add a constraint that enforces future solutions to return a higher objective value than the currently best ($y > 2$). This procedure is repeated until the optimal solution is found and optimality has been proven. Needless to say, modern solvers employ many additional strategies to improve search performance such as pre-solving or branch-and-cut. The following software packages employ, among other techniques, the above method to solve MIP problems; Gurobi (Gurobi Optimization Inc.; 2014), Xpress (*FICO Xpress*; 2016), CPLEX (IBM CPLEX; 2011), Mosek (*MOSEK*; 2016), the GNU Linear Programming Kit (*GLPK*; 2006), SCIP (Gamrath et al.; 2016), and the Coin-or branch-and-cut (Cbc) solver (Forrest et al.; 2018).

2.2.2 Constraint Programming

Constraint programming (CP) is another approach for modelling and solving combinatorial problems. Modern CP is the result of many years of research; it started with early work on constraint based systems (Sutherland; 1964), continued with work that used the PROLOG system as host framework for CP with negations (Van Emden and Kowalski; 1976), to constraint logic programming (Jaffar and Lassez; 1987) and constraint based reasoning, which introduced the general concept of a Constraint Satisfaction Problem (Guesgen and Hertzberg; 1992) and, finally, to the modern concept of Constraint Programming (Rossi et al.; 2006).

A constraint program is composed of variables, domains associated with variables, constraints and an objective function. Problems formalised using the CP approach are not limited to linear constraints and a linear objective function. To find solutions, CP solvers use a combination of propagation and search. Propagation reduces the domain of variables until no further reduction is possible or a constraint is violated. After propagation has finished, search assigns values from their domain to variables. If a constraint is violated by a variable assignment, backtracking is used to revert to previous search decisions. Modern solvers employ many additional strategies to improve the search performance.

A neat example to introduce the foundation of CP based search and propagation is the n-queens problem modelled in Fig. 2.2 and visualised in Fig. 2.3. We are given n queens

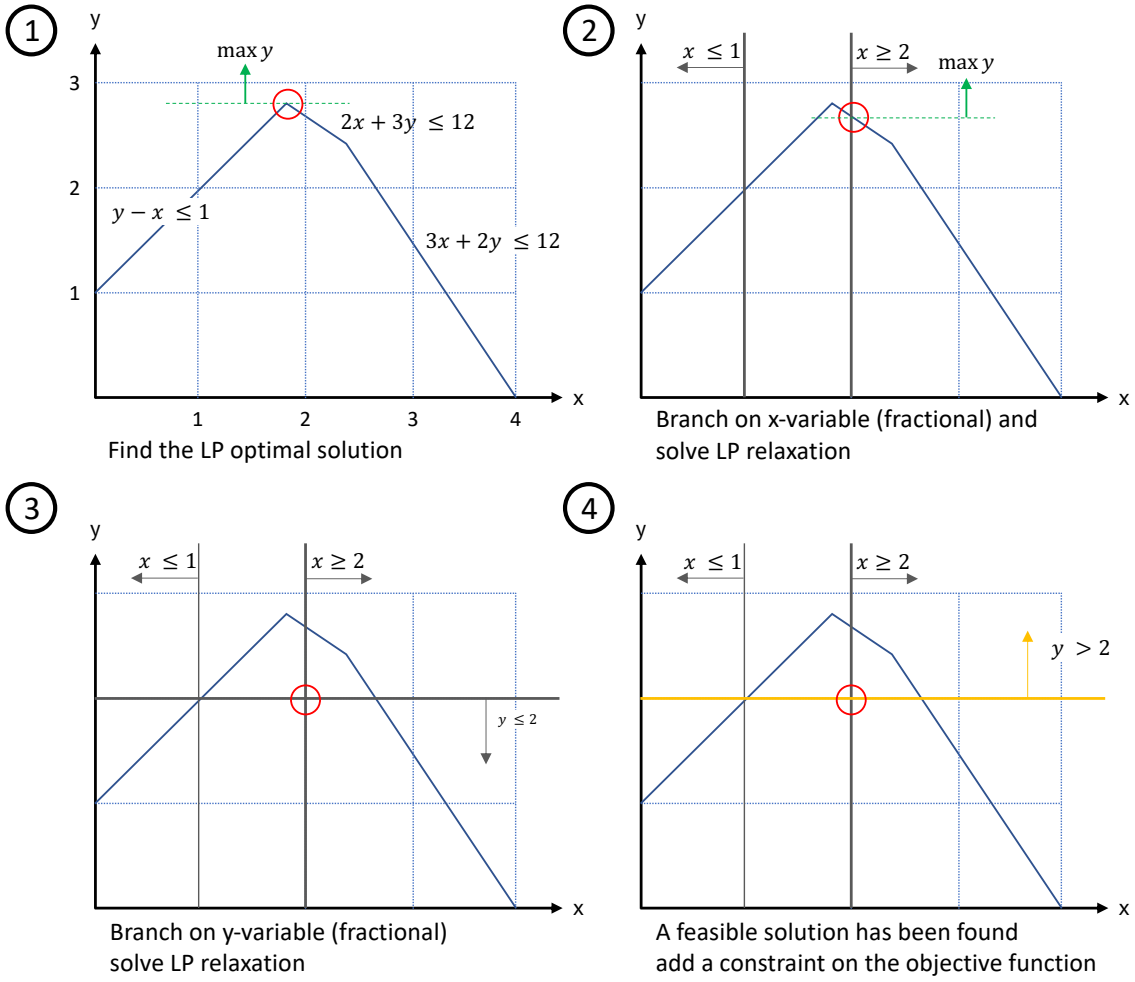


Figure 2.1: Solving a MIP problem

```
1  include "alldifferent.mzn";
2  %number of queens
3  int: n;
4  %queen of column i is placed in row q[i]
5  array[1..n] of var 1..n: q;
6
7  %distinct rows
8  constraint alldifferent(q);
9  %distinct diagonals upwards
10 constraint alldifferent([q[i] + i | i in 1..n]);
11 %distinct diagonals downwards
12 constraint alldifferent([q[i] - 1 | i in 1..n]);
13
14 solve :: int_search(q, first_fail, indomain_min, complete)
15         satisfy;
```

Figure 2.2: The n-queens problem

and a chess board of size n (n may be different than eight). The goal is to place exactly one queen in every row, column and diagonal. A possible solution is displayed in Fig. 2.3, with $n = 4$, in the bottom right corner. Each column contains exactly one queen and no two queens share a row or diagonal. The problem is modelled using variables that describe for each column where the respective queen is placed, Fig. 2.2 line 5. The domain of the variables is 1 to n; each queen is placed in a row with index $1,\ldots,n$. The `alldifferent` constraints, line 8 to 12, enforce the rules of the game.

The procedure to solve the n-queens problem is displayed in Fig. 2.3. First, we employ propagation on the empty chess board (1), top left. No domain reduction is possible at this point, as no queen has been deployed yet. Secondly, search places a queen in the first column $q[1] = 2$ (2). Propagation is now able to reduce the domain of the remaining variables (3). For example, the domain of $q[2]$ is reduced to $\{4\}$, $q[3]$ to $\{1,3\}$ and finally $q[4]$ to $\{1,3,4\}$, indicated by the red colour. Thereafter, a queen is placed in $q[2] = 4$ (the only option) and the domain of the variables is further reduced (4). Next, a queen is positioned at $q[4] = 1$, and the result of this variable assignment is a domain reduction of $q[3]$ to the empty set. This violates the constraints of the game, as one queen must be placed in every column, and therefore the current queen placement cannot yield a feasible solution. As a result, the last search decision is reversed, also called backtracking, and the queen is placed in the only alternative location (6).

Many CP solvers exist, including Gecode (Schulte et al.; 2006), ECLiPSe (Apt and Wallace; 2006), Mistral (Hebrard and Siala; 2017), Minion (Gent et al.; 2006), Opturion CPX (*Opturion CPX*; 2018), Chuffed (Chu; 2011) and Choco (Prud'homme et al.; 2014). All these solvers work on the basis of propagation and search, however employ lots of additional techniques to improve the search performance.

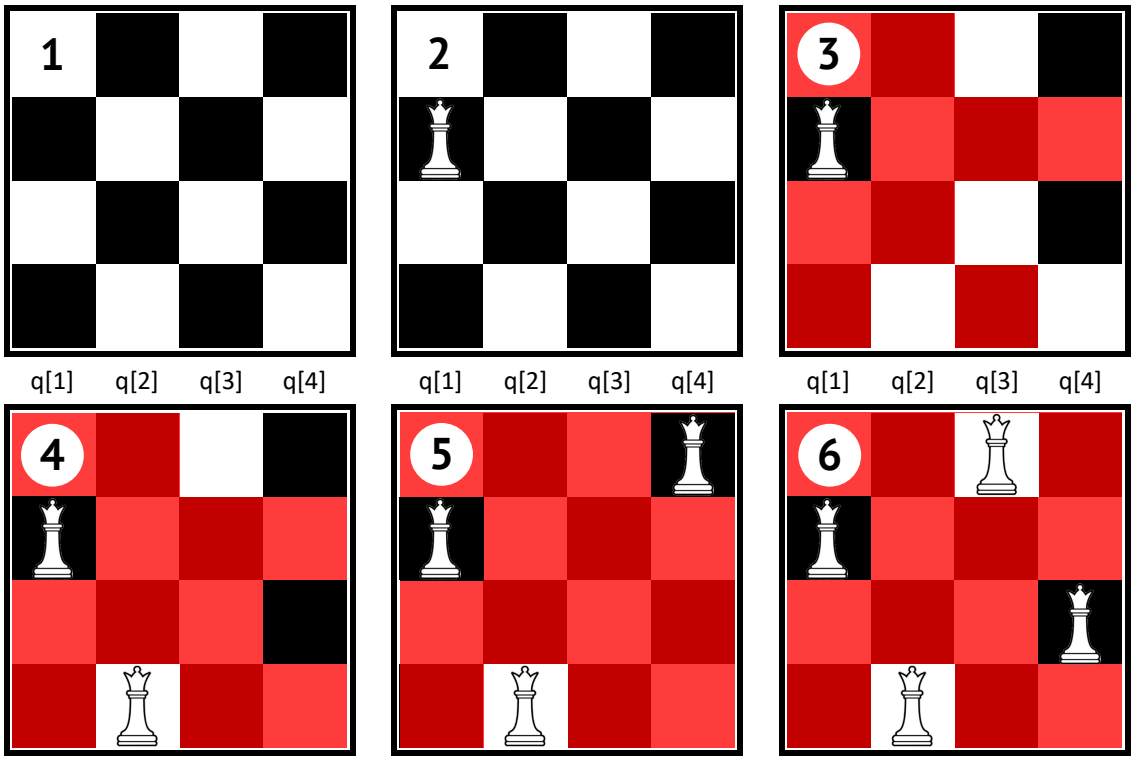


Figure 2.3: Solving the n-queens problem with Constraint Programming

2.3 Modelling Combinatorial Problems

In the previous section we introduced MIP and CP as two generic solving techniques for combinatorial problems. This section addresses how to create models of combinatorial problems that are solved by off-the-shelf (generic) solvers.

When generic solvers started to be developed, mathematical problems were typically modelled using code that would construct input for a specific solver. Later with systems based around the concept of constraint logic programming (Van Emden and Kowalski; 1976), declarative languages were used to formalise problem instances. These models could be passed on to multiple back-end solvers. In the following, we first introduce algebraic modelling before we proceed with addressing the CP modelling paradigm and conclude with contrasting both.

2.3.1 Algebraic Modelling

Algebraic modelling languages are a popular way of describing combinatorial problems. Many systems are available such as, GAMS (Bussieck and Meeraus; 2004), Pyomo (Hart et al.; 2012), AMPL (Fourer et al.; 1993), AIMMS (Bisschop and Roelofs; 2006a), Mosel (Colombani and Heipcke; 2002), and JuMP (Dunning et al.; 2017), the optimisation package for the numerical computing system Julia (Bezanson et al.; 2017). The syntax of these systems is similar to the mathematical notation of optimisation problems (Kallrath; 2013). The modelling is "solver independent", once formulated, a model can be solved using a MIP solver without customising the model for it. An algebraic model is typically

composed of continuous and discrete variables, linear constraints and a linear objective function. However, some of the systems mentioned above include support for non-linear constraints and objective function.

2.3.2 High-Level Constraint Modelling

Algebraic modelling intends to be syntactically close to the mathematical notation of optimisation problems. The CP modelling paradigm is different as it aims to capture the structure of a model on a high level of abstraction. This is achieved by using global constraints to capture common model sub-structures and by providing user-defined predicates and functions. A CP model is composed of variables, associated domains and constraints. A number of CP modelling languages are available such as, OPL (Van Hentenryck et al.; 1999), Zinc (de la Banda et al.; 2006), Essence (Frisch et al.; 2007) and MiniZinc (Nethercote et al.; 2007). Constraint models can either be solved using a CP solver directly or they can be linearised and then solved using MIP technology. In the following we will use MiniZinc to introduce the main concepts of modelling in CP by example.

2.3.3 Modelling in MiniZinc

The n-queens problem modelled in Fig. 2.3 uses MiniZinc syntax. In line 3 the parameter n is declared. For the model to become a concrete problem instance, n needs to be specified. This is usually done in a separate data file. Later, in line 5, the variables of the model are declared. A variety of variable types are available in constraint programming, such as Booleans, integers, floats, enumerated types, and strings. The types can be aggregated using arrays or sets. In line 8 to 12, the constraints of the problem are defined. As in MIP, the constraints can be linear. However, in addition, it is possible to define non-linear and logical constraints as well as global constraints, such as the `alldifferent` constraint used in the example. Global constraints enable stronger propagation as the structure of the model is explicitly captured, as opposed to using a set of disconnected constraints (Beldiceanu et al.; 2007). Lastly, the model contains a solve item, line 14. The solve item specifies that we are either interested in finding a variable assignment that satisfies all constraints or it defines whether to minimise or maximise a given objective function whilst satisfying the constraints.

CP solvers strongly rely on the search strategy used when solving a model. As a result, constraint modelling languages allow the modeller to specify a search strategy. For example, the solve item in 14 is annotated with `int_search`, in which a search strategy is specified. Understanding how search strategies work is not important to follow the rest of the thesis.

Note: While deterministic optimisation enjoys an almost universally accepted canonical form, there is no comparable standard form to express stochastic optimisation problems (Powell; 2014). MiniZinc (Nethercote et al.; 2007) and its predecessor Zinc (de la Banda et al.; 2006) are formal languages. Therefore, we use throughout the thesis the MiniZinc language to describe optimisation problems, including stochastic problems.

2.3.4 A Comparison of Algebraic and Constraint Models

We introduced the concept of algebraic modelling and the way a problem is formalised using CP. To further clarify the differences, this section will contrast models specified in either paradigm using MiniZinc notation.

Travelling salesman problem

The travelling salesman problem (Applegate et al.; 2006) is a well-studied mathematical problem. Given a road network and a set of cities, the challenge is to find the shortest path to visit each city and return to the starting location. In Fig. 2.4 the TSP is modelled using algebraic notation. The distance between each city is given by the parameter `Distance`. A two-dimensional array of variables with domain $\{0,1\}$, called `arc`, denotes whether a road segment is travelled or not. The `load` variables are used for sub-tour elimination; all cities have to be visited in one single tour. The linear constraints in line 8 to 12 state first that each city is visited exactly once followed by the sub-tour elimination constraints. The solve item indicates that we aim to find the shortest path.

The TSP can also be modelled using the CP paradigm, as shown in Fig. 2.5. Instead of using a matrix of binary `arc` variables, we define a vector of `successors` that specifies which city to visit after the current one. In the MIP formulation, a set of linear constraints was used to specify the core of the problem. When modelling in CP, the global constraint `circuit` can be used. The `circuit` constraint (Lauriere; 1978) implicitly captures the structure of the model, including sub-tour elimination.

```
1
2  array[1..N,1..N] of int: Distance;
3  array[1..N,1..N] of var 0..1: arc;
4  array[1..N] of var 0..N: load;
5
6  var int: total_distance;
7
8  constraint forall(i in 1..N)(
9    sum(j in 1..N)(arc[i,j]) = 1 /\
10   sum(j in 1..N)(arc[j,i]) = 1);
11 %Subtour elimination constraint
12 constraint forall(i,j in 2..N)(
13   load[i] - load[j] + N * arc[i,j]
14   <= N-1);
15
16
17 constraint total_distance =
18   sum(i,j in 1..N)
19   (Distance[i,j]*arc[i,j]);
20
21 solve minimize total_distance;
```

Figure 2.4: TSP in MIP

```
1  include "globals.mzn";
2  array[1..N,1..N] of int: Distance;
3
4
5  array[1..N] of var 1..N: successor;
6  var int: total_distance;
7
8
9
10
11
12
13
14
15 constraint circuit(successor);
16
17 constraint total_distance =
18   sum(i in 1..N)
19   (Distance[i,successor[i]]);
20
21 solve minimize total_distance;
```

Figure 2.5: TSP in CP

Task assignment Problem

The challenge in the task assignment problem is to assign tasks to workers and maximise the value of all completed tasks. There are at least as many tasks as workers and each worker executes exactly one task. The MIP formulation of this problem is listed in Fig. 2.6. As in the TSP, a two-dimensional array of binary variables, `w_t` is used to represent the task to worker assignment. A set of linear constraints, line 9 to 13, specifies that each worker has one task assigned (first set of constraints) and that each task is at most assigned to a single worker. The CP version of the assignment problem, Fig. 2.7, uses the global constraint `alldifferent` (Lauriere; 1978) to represent the same logic.

```
1
2  set of int: WORKER;
3  set of int: TASK;
4  array[WORKER,TASK] of int: value;
5
6  array[WORKER,TASK] of var 0..1: w_t;
7
8
9  constraint forall(w in WORKER)(
10    sum(t in TASK)(w_t[w,t] = 1));
11
12 constraint forall(t in TASK)(
13    sum(w in WORKER)(w_t[w,t] <= 1));
14
15
16 solve minimize
17    sum(w in WORKER, t in TASK)(
18    w_t[w,t] * value[w,t]);
```

Figure 2.6: Task assignment in MIP

```
1  include "globals.mzn";
2  set of int: WORKER;
3  set of int: TASK;
4  array[WORKER,TASK] of int: value;
5
6
7  array[WORKER] of var TASK: task;
8
9
10
11
12
13
14 constraint alldifferent(task);
15
16 solve minimize
17    sum(w in WORKER)(
18    value[w,task[w]]);
```

Figure 2.7: Task assignment in CP

Conclusion

In contrast to algebraic modelling frameworks that replicate mathematical syntax, the CP modelling paradigm captures the problem structure at a higher level of abstraction. Constraint Programming permits logical constraints, such as implications, conjunctions and disjunctions that introduce non-linearities into the mathematical model, which can only be expressed indirectly in algebraic models, for example with additional variables and constraints. However, each CP model can be reformulated, e.g. linearised, to match the structural properties required to be solved using MIP technology.

2.3.5 Compiling MinZinc Models

Modern high-level modelling languages, including MiniZinc, follow a three-step procedure; modelling, compilation and solving. The compile process of MiniZinc is conceptually illustrated in Fig. 2.8. The user formalises a model, that is, a parametric specification

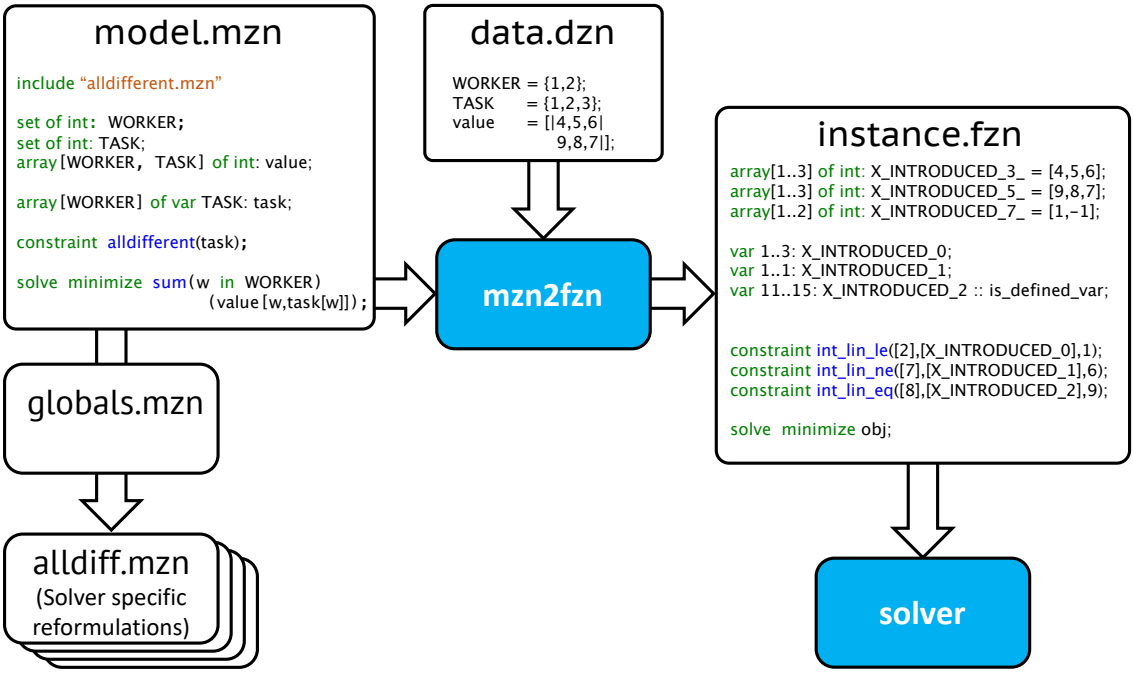


Figure 2.8: MiniZinc compilation

of the problem, top left. A model can include global constraints that are specified in a library. The compiler then creates an instance, using the parametric model, a data file and information from the library. The resulting instance is a solver specific FlatZinc file. FlatZinc contains all specifications for the solver and is not intended to be human readable.

The MiniZinc library contains solver specific information. When creating a FlatZinc instance for a solver that understands a specific global constraint, the constraint is passed on to the solver. Otherwise, the MiniZinc compiler rewrites, e.g. linearises, constraints such that a MIP solver can be used as back-end solver. The ability to reformulate the structure of a model into instances that are optimised for a specific solver makes CP modelling frameworks versatile and powerful.

2.4 Stochastic Programming

Stochastic programming is an approach for modelling and solving optimisation problems that include random variables. Many decision problems are subject to uncertainty. For example, the price of commodities, such as fuel or feed for cattle, often varies between the planning and execution stage. These fluctuations must be considered to ensure the profitability of a business. Similarly, customer demand is often unknown. While machine learning and statistical inference methods provide a means to forecast demand patterns, predictions are always subject to a confidence level that must be taken into account to implement robust decisions. Various strategies might be considered when solving a decision problem under uncertainty.

To provide an intuition for some of the basic strategies that can be employed when optimising under uncertainty, we are using an example. Consider the challenge of designing

milk pick-up routes, with unknown milk production. Once the capacity limit of the truck is met, e.g. on a high production day, the tank must be emptied before proceeding with collecting milk. One might decide to *wait-and-see* until the randomness is revealed. In our example, the trucks could be held back until the milk production for the day is known before planning the tours. However, delaying decisions is not always possible, e.g. the time to finish a tour might be long and delaying the start compromises other operations. Another approach would be to create *robust* tours that remain feasible, even in the worst case. It is possible to design milk pick-up tours that ensure enough truck capacity even when all farmers produce maximal outputs. However, such robust solutions are often conservative and may impose unnecessary cost. This thesis studies strategies where a set of decisions is implemented before observing the randomness and other decisions made after, to account for the uncertainty.

2.4.1 Two-Stage Stochastic Programming

The idea behind two-stage stochastic programming is that optimal decisions should be based on data available at the time a decision is made and should not depend on future observations. This results in two decision stages, one before observing the uncertainty and one after. To support the explanation, we use a two-stage facility location problem as illustrated in Fig. 2.9. The challenge in the classical (deterministic) facility location problem is to select a set of facilities, e.g. warehouses, that will be used to serve a number of predetermined customers, e.g. shops, whilst minimising the cost of opening facilities and distributing goods. Now, let us assume that building facilities takes time and we have to decide where to build facilities before knowing the precise customer demand. The decisions taken before observing the uncertainty are called **first-stage** decisions. For example, in the facility location problem we initially decide which facilities to open, Fig. 2.9 top left corner. **Scenarios** are used to represent the uncertainty. Each scenario captures a potential instantiation of the random variables, e.g. three scenarios are used to describe how the customer base might look like in the future. Furthermore, scenarios are associated with probabilities, e.g. not every scenario may be equally likely. In the **second stage**, decisions are taken with respect to each scenario, e.g. assigning customers to facilities. Importantly, the first-stage decisions are made without anticipating which scenario will actually occur, this is called the **nonanticipativity** principle. Finally, the goal might be to optimise with respect to the **expected** value of the problem; in the facility location example we would like to find an assignment that minimises the cost of opening the facilities (first-stage cost) and the expected cost of serving the customers (second-stage cost). The expected cost is the probability weighted cost over all scenarios, e.g. the average. Alternatively, one might seek a solution that is optimal with respect to a risk measure such as Value at Risk or Conditional Value at Risk (Jorion; 2000).

2.4.2 Multi-Stage Stochastic Programming

The last section introduced the basic concepts behind two-stage stochastic optimisation. Most importantly, it explained that decisions are split up into a first and second stage,

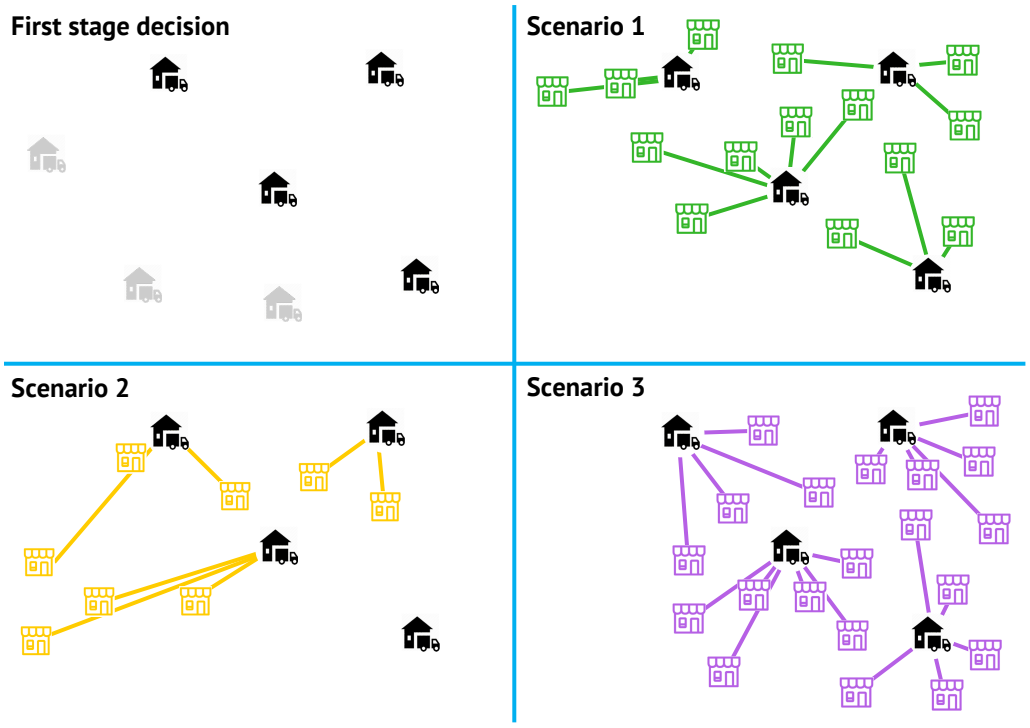


Figure 2.9: A two-stage facility location problem

where the first-stage decisions are made without anticipating the future. However, in many cases the uncertainty is revealed over time. For example, consider the management of inventory in a store. Each quarter (or other decision period), the remaining stock is assessed, and new products are ordered.

In Fig. 2.10 we illustrate a multi-stage stochastic facility location problem. In this example, we assume that customers open and close their operations over time and the layout of facilities must be adapted to meet customer demand. As in the two-stage case, a number of facilities is opened in the first stage. Thereafter, in the second stage, we assign customers to facilities. However, an additional set of decisions is made; we might decided to open further facilities to cope with future demand. In the third, and last stage, the only decision left is the assignment of customers to facilities. The scenario structure in multi-stage stochastic optimisation forms a tree. In the very first or root stage, decisions are made with respect to all scenarios. However, later, e.g. in the second stage, the scope of decisions reduces to the set of scenarios that can be reached from the current node towards the leaves, for example the second-stage decisions for scenario one and two are identical.

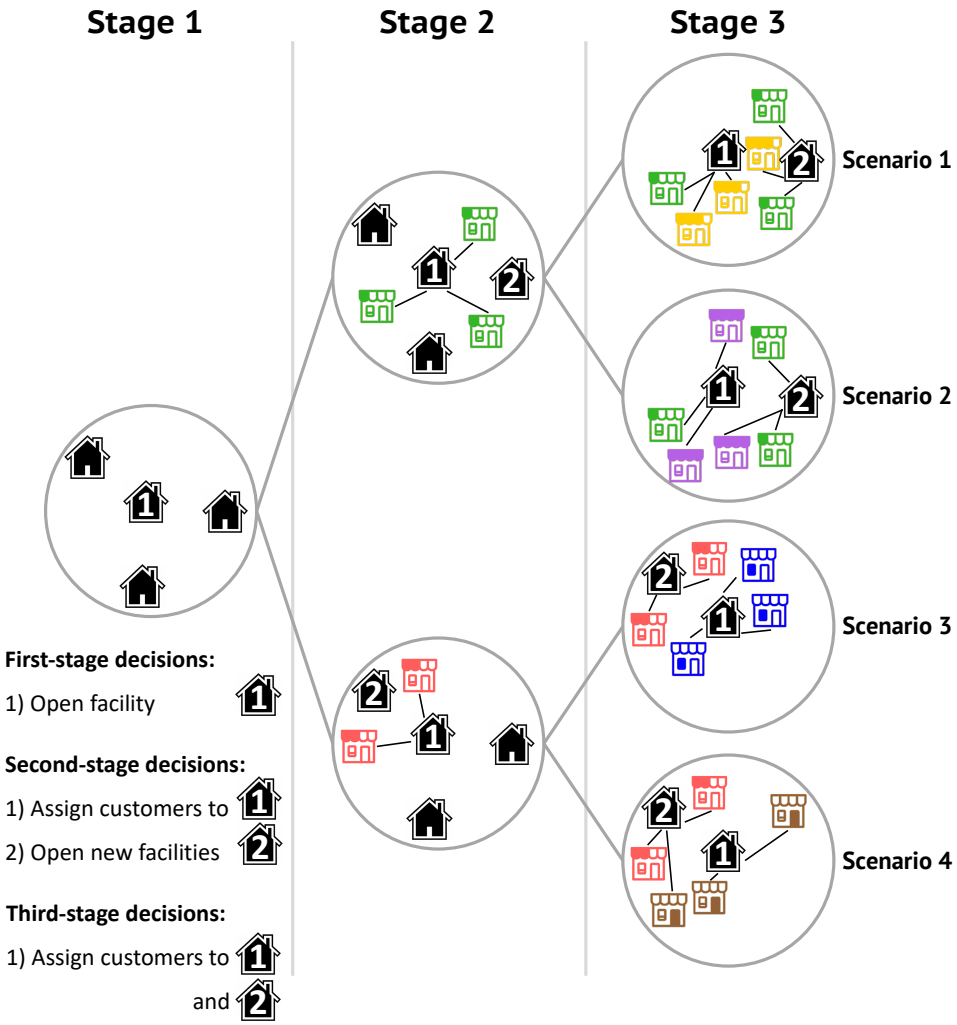


Figure 2.10: A multi-stage facility location problem

2.5 Solving Stochastic Programs

Stochastic combinatorial optimisation problems are challenging to solve and many specialised algorithms have been developed to solve them, an overview of which can be found in (Schultz; 2003). In the following we will introduce the most relevant methods to solve two-stage scenario-based stochastic problems. Many of these techniques can in theory be generalised to multi-stage problems.

Deterministic Equivalent

The most straightforward method to model and solve stochastic problems is the deterministic equivalent (DE) formulation. The DE is one large model that contains all scenarios that represent the stochastic problem. The model is composed of a set of first-stage variables and constraints. In addition, it contains copies of the second-stage variables and constraints, one for each scenario. This implies that the model grows linearly with the number of scenarios. The DE is then solved using any off-the-shelf MIP or CP solver. Using the DE is convenient as it can directly be formulated, and no specialised algorithm is required to solve it. However, for combinatorial problems, a linear increase in the size of the model (e.g. scenarios) leads to an exponential growth in the search space. As a result, deterministic equivalent models become computationally intractable when more than just few scenarios are modelled. To solve stochastic programs more effectively, many algorithms that are based on decomposition techniques have been developed.

2.5.1 Decomposition Approaches

Scenario-based stochastic optimisation problems exhibit a convenient structure to be decomposed into subproblems and solved accordingly. Decomposition approaches have been developed to exploit the model structure and are based on the idea that solving a subproblem is cheap and can therefore be done multiple times. The subproblems are coordinated by an algorithm that iteratively solves them, extracts knowledge from the solutions and distributes this knowledge across the relevant subproblems. This way, the search is progressed until the optimal solution has been found and optimality is proven. A stochastic program can be decomposed in two ways, vertically by time-stages, and horizontally by scenarios. In the following, we first address the time-stage decomposition and thereafter the scenario decomposition.

L-Shaped method

The first decomposition approach we consider is the vertical decomposition. When splitting the stochastic problem by time stages, it naturally decomposes into a master and a set of subproblems. The master problem is composed of the first stage and is augmented with an optimistic approximation of the second stage, Fig. 2.11 conceptually illustrates this. The subproblems capture the second stage. The problem is then solved using the

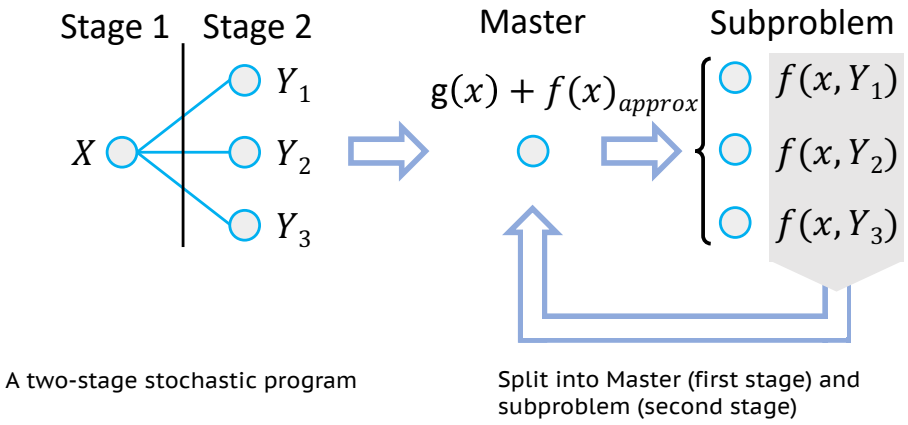


Figure 2.11: Stagewise decomposition

L-Shaped method (Birge and Louveaux; 2011), which is equivalent to Benders decomposition (Rahmaniani et al.; 2017), however called L-Shaped method in the context of Stochastic Programming.

The L-Shaped algorithm proceeds as follows. The master problem is solved and the resulting first-stage variable assignment is projected onto the subproblems before solving them. Information that is extracted from the solved subproblems is passed back into the master problem to A) eliminate infeasible variable assignments and B) guide the search towards the optimal solution. The classical L-Shaped method is restricted to problems with pure LP subproblems, as it requires the dual of the second stage. *Note, the dual of an LP consists of an alternative view on the problem that recovers information of the original problem commonly known as primal model.* The dual is widely used in algorithms, however can only be constructed for linear programs. However, Laporte and Louveaux (1993) proposed an extension to the basic algorithm; the integer L-Shaped method to solve problems with binary first-stage variables and mixed-integer linear second-stages.

2.5.2 Scenario Decomposition

The L-Shaped method is based on a time-stage decomposition. Another way to naturally decompose a Stochastic Program is horizontally by scenario. The scenario decomposition is conceptually illustrated in Fig. 2.12. A copy of the first-stage variables and constraints is introduced for each of the scenarios. In addition, a consistency constraint forcing all copies to take on the same variable assignment is introduced. When removing (or relaxing) the consistency constraints we end up with a number of separated subproblems. Algorithms that are based on the scenario decomposition iteratively enforce convergence over the first-stage variables. In the following we will introduce the most relevant methods.

Dual Decomposition

Carøe and Schultz (1999) introduced the dual decomposition (DD) to solve linear two- and multi-stage stochastic programs with integrality requirements. When using the DD, the stochastic problem is decomposed by scenarios and a branch-and-bound scheme is used to

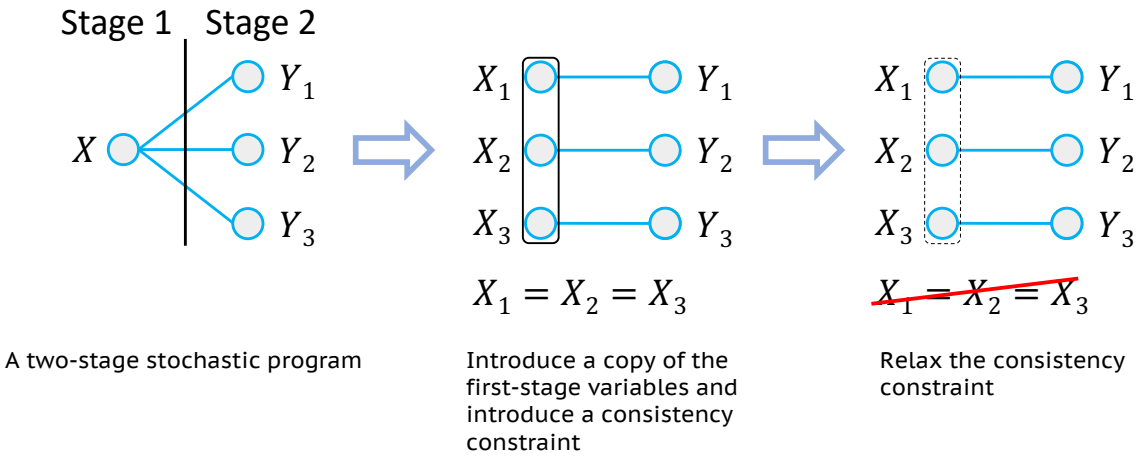


Figure 2.12: Scenario decomposition

find convergence over the first-stage variables. To obtain a lower bound, the scenarios are solved individually and Lagrangian multipliers are used to strengthen the lower bound. To get an upper bound, the average value of the first-stage variable assignments is used, but integrality is enforced first using a rounding heuristic. Subsequently, the feasible region is successively partitioned by branching over the shared (first-stage) variables. The efficiency of the DD strongly depends on the update scheme used for the Lagrangian multipliers. While Carøe and Schultz (1999) claim that the DD method is theoretically applicable to solve multi-stage problems, the authors acknowledge that due to the increased complexity, multiple issues arise to successfully implement a multi-stage version of the algorithm (Ahmed et al.; 2003).

Progressive Hedging

Rockafellar and Wets (1991) introduced Progressive Hedging (PH); a scenario decomposition based algorithm to solve convex stochastic programs. Convergence over the shared (first-stage) variables is found by successively penalising the objective function of every scenario. The algorithm first solves each scenario individually without penalty term. Thereafter, the objective function of the scenarios is augmented with a penalty and regularisation term. The scenario specific penalty is calculated using the Euclidean distance between the average overall first-stage variable assignments and the first-stage variable assignment of each individual scenario. Multiple methods that extend PH to solve problems with integrality requirements, but sacrificing optimality, have been developed (Watson and Woodruff; 2011) or (Boland et al.; 2018). Furthermore, PH can be used to solve multi-stage problems (Løkketangen and Woodruff; 1996) and (Haugen et al.; 2001).

Branch and Fix

Alonso-Ayuso et al. (2003) introduced an algorithm to solve two- and multi-stage stochastic optimisation problems, called Branch-and-Fix (BF), that relaxes the integrality requirements in addition to the consistency constraints, without introducing Lagrangian

multipliers. First, each scenario is solved using a linear programming based branch-and-bound method. Thereafter, consistency over the shared variables is regained by using a common branching mechanism that fixes the shared variables in all scenarios to identical values. From the viewpoint of the full model this corresponds to what is known as logical or constraint branching. The branching occurs step-wise until a fully converged solution is found and optimality is proven.

Scenario Decomposition for $0-1$ programs

Ahmed (2013) introduced an algorithm to solve two-stage problems with binary first-stage variables and arbitrary second-stage structure. The algorithm is based on the scenario decomposition and follows a three-step procedure. First, each scenario is solved individually. A lower bound is computed as the probability weighted sum over all scenarios. Secondly, each first-stage variable assignment, also called candidate, obtained in the first step is successively evaluated over all remaining scenarios. A valid upper bound is calculated as the probability weighted sum of the objectives that result from the evaluation step. Thirdly, all evaluated candidates are excluded from further search iterations by adding a nogood constraint to each of the scenarios. By iteratively repeating the three steps the algorithm is guaranteed to find the optimal solution.

2.6 Stochastic Programming Frameworks

Stochastic Programming is a popular framework for decision making under uncertainty (Valente et al.; 2009). While it is possible to formalise stochastic programs in standard algebraic and CP modelling frameworks, as introduced in section 2.3, there are numerous drawbacks when attempting to do this. First, the models become large and unclear. The decisions, uncertainty and stages are all expressed in the same model. This can be confusing and changing the model becomes cumbersome. Secondly, due to the special structure of scenario-based stochastic programs, algorithms that exploit the structure have been developed with great success. When using a single model to express the decisions and uncertainty, it becomes difficult to utilise these special solving methods. Many extensions to standard modelling frameworks have been proposed to address the mentioned drawbacks. In the following, we will first introduce the idea behind these extensions before addressing algorithms to solve these problems.

2.6.1 Modelling

Stochastic Programming extensions to classical modelling frameworks aim to increase the usability of Stochastic Programming for optimisation practitioners. While there are many nuances when comparing the frameworks, the basic idea remains conceptually the same. An illustration of this concept is shown in Fig. 2.13. As introduced earlier, modern modelling languages are based on the idea of formalising a parametric model of the optimisation problem. The parametric model combined with a set of data is then compiled into a non-parametric instance that is passed on to a solver. The stochastic extensions

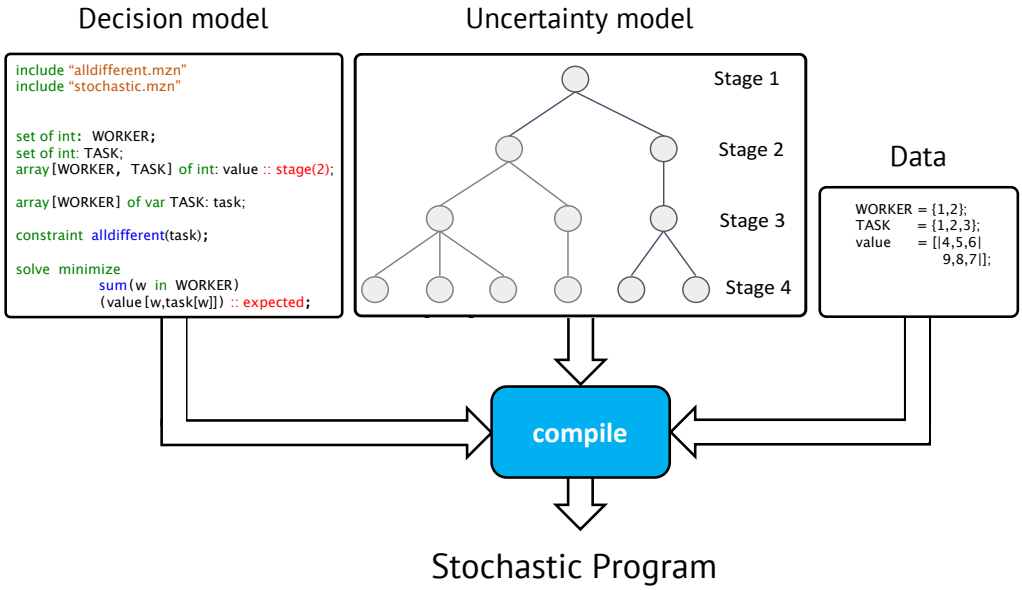


Figure 2.13: A stochastic program modelled in a high level language

adopt this idea. A decision model, left hand side in Fig. 2.13, describes a deterministic optimisation problem. Instead of adapting the decision model to take on the form of a stochastic problem, a parametric meta model is used to characterise the uncertainty. The meta model assigns variables and parameters to stages, describes the objective function, e.g. maximising the expected value, and specifies whether a constraint must be satisfied in all scenarios (a hard-constraint) or satisfied with a given probability (a chance-constraint). The parametric decision and uncertainty model is then compiled together with a set of data into a concrete stochastic programming instance. This concept allows the compiler to produce multiple output models, depending on the solving approach used to find solutions.

2.6.2 Stochastic Extensions to Algebraic Modelling Systems

This section will introduce the most relevant stochastic extensions to algebraic modelling frameworks.

Stochastic Programming in AIMMS

From any deterministic linear or mixed-integer model formulated in AIMMS (Bisschop and Roelofs; 2006a), the AIMMS compiler is able to automatically create a stochastic model, without the need to reformulate any of the constraint definitions (Bisschop and Roelofs; 2006b). There are two steps necessary to create a stochastic model. First, the user indicates which parameters and variables in the deterministic model shall become stochastic and secondly, the user provides a scenario tree and stochastic data. Once the stochastic model is formulated, AIMMS supports two solving methods. First, it is able to compile the model into the deterministic equivalent that is solved using a MIP solver. Secondly, for purely linear mathematical problems, an implementation of the L-Shaped method is available.

Stochastic Programming in AMPL

Valente et al. (2009) describe how stochastic programs can be expressed using AMPL. They propose to overlay the deterministic (core) model with a description of the random values and stage structure, essentially a meta model. It is possible to define two- and multi-stage stochastic MIP problems with or without chance-constraints. These models are either transformed into the SMPS format (Gassmann; 2005), a standard format for stochastic linear programs, and solved by a solver that can process the SMPS, e.g. (Ellison et al.; 2010) or solved directly as a deterministic equivalent.

Stochastic Programming in GAMS

In *GAMS - A User's Guide* (2017) it is described how a deterministic GAMS model can be augmented to express stochastic programs using the GAMS Extended Mathematical Programming (EMP) package. It is possible to define two- and multi-stage MIP stochastic programs with chance constraints. Furthermore, it is possible to define an objective function that optimises either with respect to the expected value, or a risk measure such as Value at Risk (VaR) or Conditional Value at Risk (CVaR). The models can be compiled into a deterministic equivalent and solved by a standard MIP solver. Alternatively, the DECIS (Dantzig and Infanger; 1991) or LINDO (*GAMS - A User's Guide*; 2017) solver can be used to solve two-stage linear programs using Benders decomposition.

Stochastic Programming in Pyomo

Watson et al. (2012) propose PySP, a stochastic extension to the algebraic modelling package Pyomo (Hart et al.; 2012), a package to model optimisation problems in Python. To formulate a stochastic program in PySP, the user specifies both the deterministic base model (supporting linear, non-linear, and mixed-integer components) and the scenario tree model. Given these two models, PySP provides two options for solving the corresponding stochastic program. First, it can be compiled into a deterministic equivalent and solved using a standard MIP solver. Secondly, Watson et al. (2012) propose to use Progressive Hedging as an alternative.

2.6.3 Stochastic Extensions to Constraint Modelling Systems

Similar extensions to express stochastic programs have been made to constraint modelling systems.

Stochastic OPL

Walsh (2002b) proposed Stochastic OPL, an extension to OPL (Van Hentenryck et al.; 1999), as a framework to formalise Stochastic Constraint Programs. It extends Constraint Programming with stochastic variables, chance constraints and optimises over expectations. Conceptually this proposal follows what has been introduced earlier, a core model is augmented using a meta model. The model is then compiled into a deterministic equivalent. It is proposed to use CP technology to solve the resulting instance. A working

version of Stochastic OPL cannot be found online and Rendl et al. (2014) confirm that Stochastic OPL has never been made available.

Stochastic MiniZinc

Rendl et al. (2014) introduced Stochastic MiniZinc, an extension to the constraint programming framework MiniZinc introduced earlier. We will introduce Stochastic MiniZinc in more depth as it is fundamental for subsequent chapters. The design of Stochastic MiniZinc follows four objectives. (1) The extension is *conservative*, the stochastic model can be run, debugged, and solved deterministically without changing the core model. (2) In line with standard MiniZinc, the model remains agnostic of the solving approach. There is no need for the user to commit to a specific solving approach during the modelling. (3) The target audience of Stochastic MiniZinc are optimisation practitioners without deep knowledge of Stochastic Programming. (4) The extensions are lightweight additions to the language.

A Stochastic MiniZinc problem specification is composed of three parts: a core model, a deterministic and a stochastic data specification. An example is displayed in Fig. 2.14, where we reuse the task assignment introduced in Fig. 2.7. The core model is based on the standard deterministic model formulation. However, in addition it is annotated with `:: stage(n)` and `:: expected` information. The stage annotation associates variables and parameters with stage n. Variables and parameters without annotations belong per default to the first stage. The objective function is annotated with `expected` to express the intention to optimise the expected value over all scenarios. The deterministic data file contains all the data that is known a priori, and the stochastic data file describes the scenarios and their likelihood of occurrence. In the stochastic task assignment example, three scenarios are used to describe the random variable `value`. The `weights` denote how likely a scenario will come true, e.g. the second scenario is three times as likely to occur compared to scenario one and three. The weights in the description do not add up to 1 (as one would expect for probabilities), however when normalising the weights, the result is equivalent to using probabilities in the first place.

A Stochastic MiniZinc model can be compiled into a deterministic equivalent and either be solved using a CP solver or it can be linearised and passed on to a MIP solver. In addition, Rendl et al. (2014) propose to use Combinators (Schrijvers et al.; 2013) or MiniSearch (Rendl et al.; 2015) to implement either Progressive Hedging or policy-based search (Walsh; 2002a) to solve the problems.

```
%core_model.mzn
include "globals.mzn";
include "stochastic.mzn";

set of int: WORKER;
set of int: TASK;
array[WORKER,TASK] of int: value  ::  stage(2);

array[WORKER] of var TASK: task ::  stage(1);

constraint alldifferent(task);

solve minimize sum(w in WORKER)(value[w,task[w]]) ::  expected;
```

```
%deterministic_data.dzn
WORKER = {1,2};
TASK = {1,2,3};
```

```
%stochastic_data.sdzn
value = array3d(1..3,1..2,1..3,
          [4,5,6, %scenario 1
           3,4,2,
           3,2,1, %scenario 2
           6,7,9,
           9,4,3, %scenario 3
           8,2,1]);
array[1..2] of int: weights[1,3,1]
              ::  scenario_weights;
```

Figure 2.14: Stochastic task assignment in Stochastic MiniZinc

2.7 Formalism

Thus far, this chapter gave a high-level overview of the main concepts that are relevant for the rest of the thesis. The purpose of this section is to lay the formal foundations used throughout the rest of the thesis. We start by introducing the notation of a Constraint Optimisation Problem, before moving on to the formalities of a Stochastic Program. The section is then closed with a more in-depth discussion of the scenario decomposition introduced in section 2.5.2.

2.7.1 Constraint Optimisation Problem

A deterministic Constraint Optimisation Problem is defined as follows:

Definition 1 *A **constraint optimisation problem** (COP) is a four-tuple $\mathcal{P}_\mathcal{D}$:*

$$\mathcal{P}_\mathcal{D} = < V, D, C, f >$$

where V is a set of decision variables, D is a function mapping each element of V to a domain of potential values, and C is a set of constraints. A constraint $c \in C$ acts on variables $x_i, \ldots, x_j$, termed scope(c) and specifies mutually-compatible variable assignments σ from the Cartesian product $D(x_i) \times \cdots \times D(x_j)$. The quality of a solution is measured using the objective function f. We write scope(σ) for the variables that appear in σ; $\sigma(x)$ for the value of x in assignment σ; $\sigma|_X$ for σ restricted to the set of variables X; and $\sigma \in D$ means $\forall x : \sigma(x) \in D(x)$. We write the union of two assignments (with disjoint

scopes) $\sigma_1 \wedge \sigma_2$*. Furthermore, we define the set of* ***solutions*** *of a COP as the set of assignments to decision variables from the domain* D *that satisfy all constraints in* C*:*

$$sol(\mathcal{P}_{\mathcal{D}}) \quad = \{\sigma \in D \mid \forall c \in C : \sigma|_{scope(c)} \in c\}$$

Finally, an ***optimal solution*** *is one that minimises the objective function:*

$$\underset{\sigma \in sol(\mathcal{P}_{\mathcal{D}})}{\arg\min} f(\sigma)$$

2.7.2 Stochastic Constraint Satisfaction Problem

The seminal paper *Stochastic Constraint Programming*, written by Walsh (2002a), defines a Stochastic Constraint Satisfaction Problem.

Definition 2 *A* ***Stochastic Constraint Satisfaction Problem*** *(SCSP) is a six-tuple:*

$$< V, S, D, C, P, \theta >$$

where S *is a subset of* V *that describes the random variables.* P *is a mapping from* S *to probability distributions for the domains;* $P(S) : S \rightarrow (\mathbb{R}^{|S|} \rightarrow [0,1])$*. The subset* $h \in C$ *that constrains at least one variable in* S *are called chance constraints. Chance constraints must be satisfied with probability* θ*, which lies in the interval* $[0,1]$*; a chance constraint with* $\theta = 1$ *is equivalent to a hard constraint.*

A SCSP is composed of one or multiple stages. In a one-stage SCSP, values are assigned to the decision variables V before observing the stochastic variables S. No decisions are made once the random variables are revealed. In a multi-stage SCSP, V and S are partitioned into disjoint sets that match the decision stages. In the case of an $m-$stage SCSP, V and S are partitioned into the disjoint sets $V_1, \ldots V_m$ and $S_1 \ldots S_m$. A feasible solution satisfies all hard constrains and violates no more than $1-\theta$ of the chance constrains.

Walsh (2002a) defines a **policy** is a tree with two types of nodes, decision variables and stochastic variables, as displayed in Fig. 2.15. Nodes that represent decision variables have a single child, whilst a stochastic variable node has one child for every possible (discrete) value it can take on. A path through the policy tree starts at the root node, which is by definition a decision variable, and ends at a leaf. Each path represents a possible world, is associated with a probability of occurrence and its leaf is labelled with either 1, if the variable assignments along the path satisfy all constraints or otherwise 0. The probability weighted sum of the leaf labels must exceed θ for the policy to be feasible.

2.7.3 Scenario based Stochastic Constraint Optimisation Problem

The semantics used by Walsh (2002a) to define a SCSP produces AND/OR trees, Fig. 2.15. Decision variables produce OR nodes where a single satisfying assignment is required (only one arc leaves the node) to satisfy the stochastic variables that form AND nodes (multiple

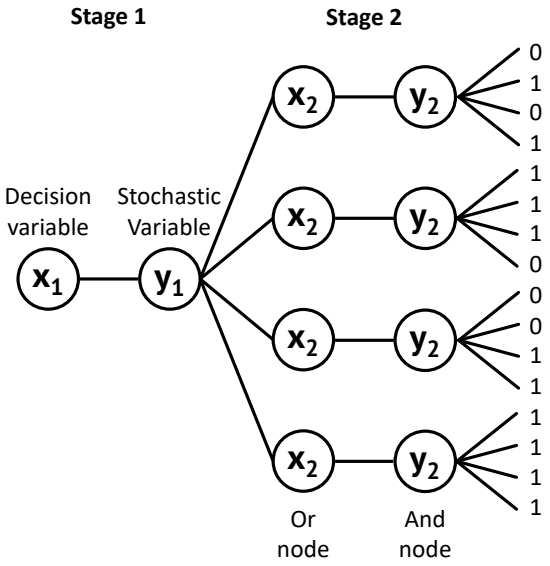


Figure 2.15: AND/OR tree

arcs). Walsh (2002a) introduces a backtracking and forward checking algorithm to find satisfiable policy trees.

Tarim et al. (2006) proposed to use an alternative semantics, the scenarios-based view, where all possible realisations of the random variables are modelled explicitly in a scenario tree as in Fig. 2.16 on the left side. The nodes represent decision variables and the arcs concrete instantiations of the random variables. Tarim et al. (2006) argue that in contrast to the policy-based view, a problem modelled using the scenario-based approach can directly be compiled into a standard (non-stochastic) constraint program - the deterministic equivalent. We will address and adopt the scenario representation for the rest of this section.

This thesis only addresses a subset of problems covered by the SCSP notation, namely problems without chance-constraints but an objective function. As a result, we leave out chance constraints and explicitly introduce a utility term. Using similar notation to Walsh (2002a), we could formalise this as a six-tuple $< V, S, D, C, P, G >$ that omits θ and introduces G, the utility function. As in Walsh (2002a) and Tarim et al. (2006), V and S implicitly contain the stage structure. Next we show how the definition of a scenario-based stochastic COP can be compiled into a form that explicitly states the stage structure.

Note: Other works have used various notations in the context of stochastic Constraint Programming. Babaki et al. (2017) explicitly express an ordering of the variables V and S in their problem definition of a *Factored Stochastic Constraint Problem* by introducing $\prec$ to end up with $< V, S, D, C, P, f, \prec >$, where $\prec$ denotes a partial ordering, e.g. $V_1 \prec S_1 \prec \ldots V_m \prec S_m$. Furthermore, Rossi et al. (2015) introduced a SCSP that states the stage structure explicitly as $< V, S, D, P, C, \beta, L >$, with $L = [< V_1, S_1 >, \ldots, < V_i, S_i >, \ldots, < V_m, S_m >]$ for m stages. We introduce a notation that differs from the formalism introduced by Babaki et al. (2017) and Rossi et al. (2015). We do this because the algorithmic concepts that are discussed in this thesis are substantially different to their work.

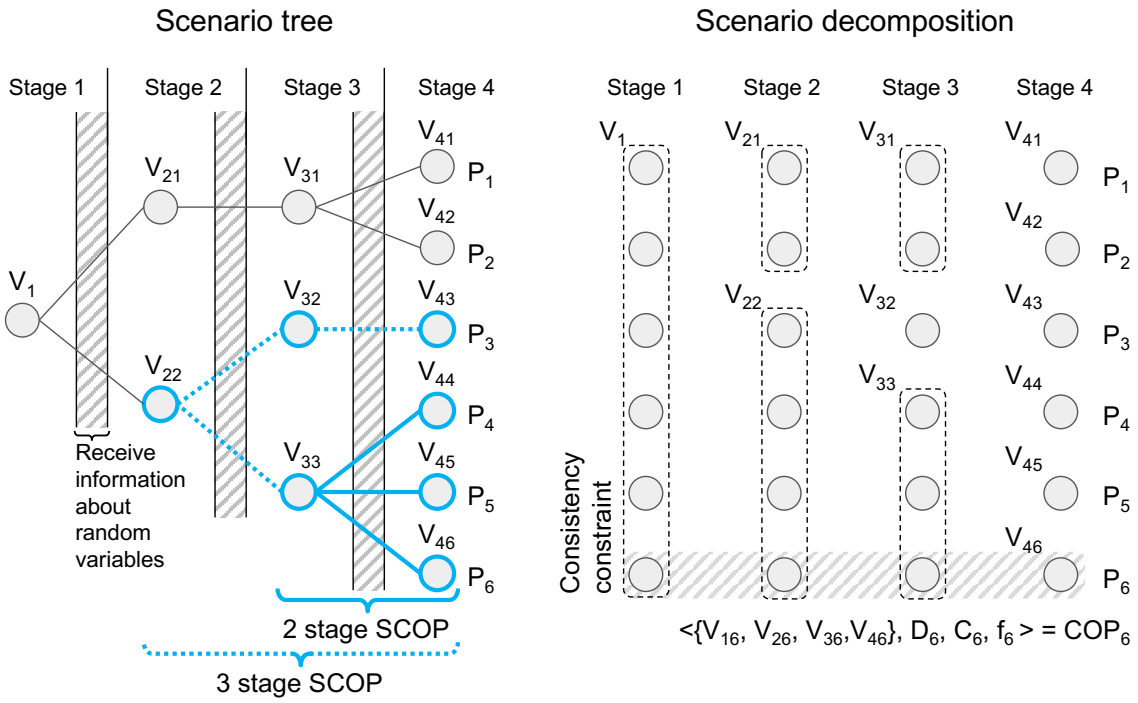


Figure 2.16: Scenario tree

Definition 3 *A **stochastic constraint optimisation problem** (SCOP) is a pair:*

$$\begin{aligned} \mathcal{P}_{\mathcal{S}} &= < V, [\mathcal{P}_{\mathcal{S}1}, \ldots, \mathcal{P}_{\mathcal{S}n}] > \\ &\quad \textit{or a COP} \\ \mathcal{P}_{\mathcal{D}i} &= < V_i, D_i, C_i, f_i > \qquad \forall i \in 1 \ldots k : V \subseteq V_i \end{aligned}$$

where $\mathcal{P}_{\mathcal{S}}$ is defined as a tree composed of the subproblems $\mathcal{P}_{\mathcal{S}1} \ldots \mathcal{P}_{\mathcal{S}n}$ and their shared variables V. Each leaf $\mathcal{P}_{\mathcal{D}i}$ is a COP that represents a scenario $i \in 1, \ldots, k$. The variables V_i of each $\mathcal{P}_{\mathcal{D}I}$ are the union of all shared variables V from the root to each leaf, including the leaf variables.

In the case of a two-stage SCOP the problem is composed of one set of shared variables V and k subproblems, $< V, [\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}k}] >$.

The multi-stage case is displayed on the left side of Fig. 2.16 using a four-stage SCOP. The illustration consists of six scenarios, where each scenario denotes a path from the root to a leaf node. Note that a subset of scenarios shares parts of a path. For example, the paths of scenarios $\mathcal{P}_{\mathcal{D}1}$ and $\mathcal{P}_{\mathcal{D}2}$ only diverge after stage three but differ from scenario $\mathcal{P}_{\mathcal{D}3}$–$\mathcal{P}_{\mathcal{D}6}$ already after stage one. This implies that the decisions taken in stage one are consistent across all scenarios, yet decisions made in stage two and three must only be consistent over scenario $\mathcal{P}_{\mathcal{D}1}$ and $\mathcal{P}_{\mathcal{D}2}$, and likewise for scenarios $\mathcal{P}_{\mathcal{D}3}$–$\mathcal{P}_{\mathcal{D}6}$. We call a solution that satisfies all constraints in the SCOP a policy tree.

Definition 4 *A **policy tree** $\mathcal{T}$ is defined as*

$$\mathcal{T} =< \sigma, [\mathcal{T}_1, \ldots, \mathcal{T}_n] >$$

*and contains an assignment of the root-stage variables σ and a list of policy trees $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$, one for each branch in the scenario tree. A policy tree $\mathcal{T}$ **matches** an SCOP $\mathcal{P}_\mathcal{S}$, if and only if the scope of each assignment σ matches the variables of the corresponding scenario and stage:*
*$\mathcal{T} =< \sigma, [\mathcal{T}_1, \ldots, \mathcal{T}_n] >$ matches $\mathcal{P}_\mathcal{S} =< V, [\mathcal{P}_{\mathcal{S}1}, \ldots, \mathcal{P}_{\mathcal{S}m}] >$ iff $scope(\sigma) = V$ with $n = m$, and for all $i \in 1 \ldots n$, $\mathcal{T}_i$ matches $\mathcal{P}_{\mathcal{S}i}$. A **path** of a policy tree $\mathcal{T}$ and a matching $\mathcal{P}_\mathcal{S}$ is a tuple $< [\sigma_1, \ldots, \sigma_d], \mathcal{P}_\mathcal{D} >$, collecting all the assignments σ_i from the root to a leaf of $\mathcal{T}$, and the COP $\mathcal{P}_\mathcal{D}$ at the corresponding leaf of $\mathcal{P}_\mathcal{S}$. We write $paths(\mathcal{T}, \mathcal{P}_\mathcal{S})$ for this set of paths. Finally, the set of **solutions** of a SCOP $\mathcal{P}_\mathcal{S}$ is defined as the matching policy trees for which each path is a solution to the underlying COP $\mathcal{P}_\mathcal{D}$:*

$$\begin{aligned} sol(\mathcal{P}_\mathcal{S}) = \{ \quad & \mathcal{T} \mid \mathcal{T} \text{ matches } \mathcal{P}_\mathcal{S}, \\ & \forall \text{ paths } p = < [\sigma_1, \ldots, \sigma_d], \mathcal{P}_\mathcal{D} > \\ & \in paths(\mathcal{T}, \mathcal{P}_\mathcal{S}) : \sigma_1 \wedge \cdots \wedge \sigma_d \in sol(\mathcal{P}_\mathcal{D}) \} \end{aligned}$$

*And finally, an **optimal solution** to an SCOP minimises the sum of the individual objectives:*

$$G = \underset{\mathcal{T} \in sol(\mathcal{P}_\mathcal{S})}{\arg\min} \quad \sum_{<[\sigma_1,\ldots,\sigma_d],<V,D,C,f>> \in paths(\mathcal{T},\mathcal{P}_\mathcal{S})} f(\sigma_1 \wedge \cdots \wedge \sigma_d)$$

Each scenario COP is associated with a probability of occurrence. In the case where all scenarios are equally likely, the expected value is simply the sum over all scenario objectives divided by the number of scenarios. In all cases with non-uniform probability distributions the contribution of each scenario objective to the expected value is weighted by its probability. We decided to use a notation that contains the scenario probabilities implicitly in each f_i, to simplify the presentation of subsequent algorithms. However, this design choice does not change the generality of the methods discussed later.

To provide an intuition of how a deterministic equivalent formulation of an SCOP can be modelled differently, we introduce an alternative definition of an SCOP that matches the above definitions, including policy tree and optimal solution.

Definition 5 *A **deterministic equivalent (DE)** can also be defined as the compact representation of an SCOP:*

$$\mathcal{P}_{\mathcal{S}DE} =< \bigcup_{i \in K} V_i, \bigcup_{i \in K} D_i, \bigcup_{i \in K} C_i, \sum_{i \in K} f_i(x) >, \tag{2.2}$$

with k_i, V_i, D_i, C_i *and* f_i matching the scenarios, variables, domains, constraints and objective functions used in the previous definitions. The DE is essentially one large model that contains all variables and constraints required to describe the SCOP.

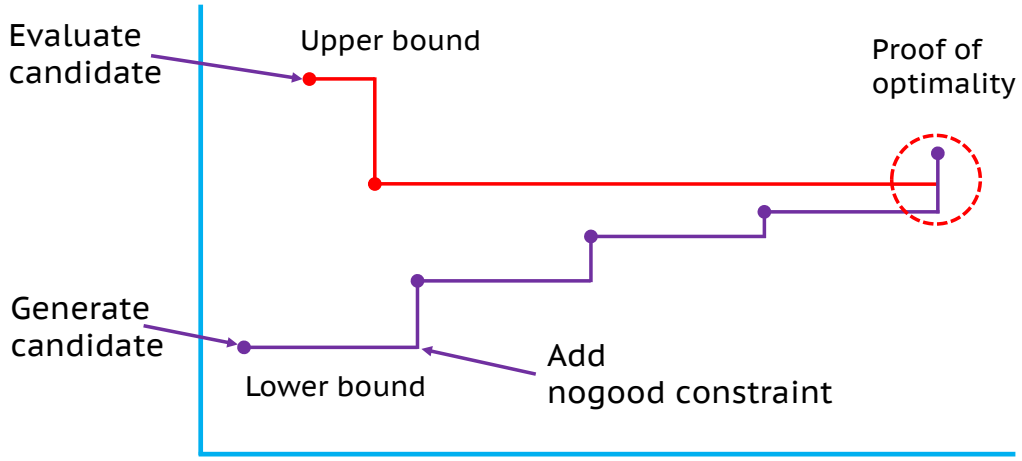


Figure 2.17: Evaluate and Cut convergence

2.7.4 Evaluate and Cut

In section 2.5.2 we introduced an algorithm to solve two-stage stochastic programs called Scenario Decomposition for 0–1 Programs (Ahmed; 2013). We now examine the Scenario Decomposition algorithm more closely as it provides the foundation for three subsequent chapters. As its name reveals, the algorithm is based on the scenario decomposition approach, where a copy of the first-stage variables is introduced for each scenario. Conceptually, the algorithm is composed of three steps; Fig. 2.17 supports the explanation. First, each scenario is solved individually, without enforcing the consistency constraints. The probability weighted sum of all objectives is a lower bound (given we are minimising), as displayed in Fig. 2.17 in the bottom left corner. Secondly, the first-stage variable assignments of each scenario, also called candidates, are evaluated across all scenarios to obtain an upper bound, as in Fig. 2.17 in the top left corner. And lastly, the explored candidates are excluded from the subsequent search, by making the corresponding variable assignment infeasible using nogood constraints. By iteratively repeating the three steps, the optimal solution is found and proven, once the (monotonically increasing) lower bound and the upper bound (incumbent objective value) meet. For convenience we rename the Scenario Decomposition for 0-1 Programs to **Evaluate and Cut** (E&C) and will refer to the algorithm for the rest of the thesis as such.

Algorithm

Algorithm 1 explains in pseudo code how SCOPs are solved using E&C. The description omits probabilities, as we assume that the each scenario COP implicitly contains the probability in its objective function. This is done to simplify the presentation, however does not compromise the generality of the method. The algorithm works for problems with uniform and non-uniform probabilities.

First, the individual scenario COPs are retrieved from the scenario tree (line 3). Each COP is then solved individually using a standard MIP or CP solver (line 8). The solution σ denotes an assignment to the first- and second-stage variables and the sum of the scenario

objectives (obj) yields a lower bound on the SCOP. The first-stage assignments σ_V of each scenario COP, called *candidates*, are *evaluated* against all other scenarios by projecting their variable assignment onto the first-stage variables of the other scenarios (line 7). Adding up the (probability weighted) objectives that result from the candidate evaluation yields an upper bound. Finally, the evaluated first-stage assignment σ_V is cut (pruned) from the search by adding a *nogood* constraint to each of the scenario COPs (line 17). By iteratively repeating the three steps – obtaining, evaluating and cutting candidates – the procedure is guaranteed to find an optimal solution, as long as the first-stage variables have finite domains. Completeness holds because the lower bound is monotonically increasing as a result of cutting off the evaluated solutions. Optimality is proven once the lower bound meets the upper bound. The scenarios can be evaluated independently, allowing highly parallelized implementations.

E&C has been successfully used to solve two-stage stochastic programs with binary first-stage variables. Basçiftci et al. (2017) solve a generator maintenance and operations scheduling problem under uncertain failure times. In their experiments they confirm the effectiveness of E&C, furthermore they introduce a stronger problem specific nogood cut that improves performance. Lei et al. (2016) use E&C to solve a pre-positioning and real-time allocation problem for resilient response to natural disasters. Ryan, Rajan and Ahmed (2016) propose a series of algorithmic improvements, such as solving the LP relaxation of the subproblems and only solving the non-relaxed subproblem if the result of the relaxation seem promising. Furthermore, they introduced an asynchronous parallelisation scheme that alleviates the need to wait at every iteration for all the scenarios to be solved. In, (Ahmed et al.; 2017) and (Deng et al.; 2017), the algorithm is used as upper bounding procedure for chance constraint programs.

Limitations

Numerous studies have shown the effectiveness of E&C, however the method has limitations. First, E&C works well for problems with binary first-stage variables, however it cannot solve problems with continuous first-stage variables. Secondly, experiments have shown that the algorithm finds the optimal solution rather quickly, often within the first few iterations. However, proving optimality can be lengthy, as the nogood cuts added in each iteration are weak, each excluding only one solution at a time. In chapter 4, we address this limitation by proposing a method that finds nogood constraints that prune more strongly compared to the candidate nogoods used in standard E&C. Thirdly, E&C is an algorithm to solve two-stage stochastic problems; however as introduced earlier, in certain circumstances, stochastic programs are modelled using multiple stages. In chapter 5, we develop a multi-stage version of E&C.

Algorithm 1 Evaluate and Cut for Two-Stage Problems

```
1: procedure SOLVESCOP(P_S)
2:     Initialize: UB = ∞, LB = -∞, sol = null
3:     [P_D1,...,P_Dk] = GET_SCENARIOS(P_S)
4:     while LB < UB do
5:         LB = 0, S= ∅
6:         % Find first-stage candidates (lower bound)
7:         for i in 1..k do
8:             < σ,obj> = SOLVE(P_Di)
9:             LB += obj
10:            S ∪ = {σ|_V}
11:        % Evaluate first-stage candidates (upper bound)
12:        for σ_V ∈ S do
13:            t_UB = 0
14:            for i in 1..k do
15:                < _,obj> = SOLVE(P_Di[C ∪ = {σ_V}])
16:                t_UB += obj
17:                ADDNOGOOD(P_Dk,σ_V)
18:            if t_ub < UB then
19:                sol = σ_V
20:                UB = t_UB
21:    return sol
```

2.8 Conclusion

In this chapter, we introduced the most relevant concepts required to understand the rest of the thesis. First, we introduced what combinatorial optimisation is and how to solve these types of problems. We made a distinction between CP and MIP modelling and solving techniques. The difference is important as subsequent chapters will utilise CP as well as MIP solvers to solve subproblems. Depending on the instance used for the experiments, one or the other technology will be more suitable. In addition, we introduced two- and multi-stage stochastic programming, relevant modelling frameworks and algorithms to solve stochastic programs.

The main take away messages of this chapter are the following. First, solving stochastic combinatorial optimisation problems is difficult and specialised algorithms are required, to tackle problems with more than just a few scenarios. Secondly, a number of modelling frameworks have been developed to formalise stochastic programs and solve them either with a standard MIP or CP solver technology or by using specialised back-end solvers. However, most of these systems are built on top of algebraic modelling frameworks and do not incorporate the paradigm and progress made in constraint programming.

This thesis will examine how to advance stochastic programming modelling frameworks that are based on the CP modelling paradigm. However, all methods we develop are equally applicable to modelling frameworks that are build on top of algebraic modelling systems.

Chapter 3

Modelling

3.1 Introduction

Stochastic programming is a powerful tool for decision making under uncertainty. To enable non-specialists the use of stochastic programming, two factors must be addressed (Watson et al.; 2012). First, it should easily be possible to express, i.e. model, stochastic programs in a similar fashion to traditional deterministic optimisation problems. Secondly, as solving stochastic programs is computationally hard, technologies that effectively exploit the underlying structure of stochastic problems must be available. This chapter is concerned with the modelling aspect of stochastic programming.

Multiple modelling frameworks that support optimisation practitioners to formulate stochastic problems have been proposed in the past. Tarim et al. (2006) implemented a framework on top of the OPL constraint modelling language (Van Hentenryck; 1999). A standard OPL model consists of declarations and instructions. Declarations define data types, constants and decision variables. Instructions on the other hand, define whether the user seeks a variable assignment that simply satisfies the constraints or is interested in finding the optimal solution which is measured by an objective function. Tarim et al. (2006) extended the declarations of OPL to include stochastic variables, instructions to define chance constraints, and a range of objectives, such as minimising the expectation. Watson et al. (2012) proposed PySP, an extension of the Pyomo open-source algebraic modelling language (Hart et al.; 2012). To formulate a stochastic program in PySP, the user specifies a deterministic base model, essentially a single scenario, and a scenario tree that contains the uncertain parameters. The model is then automatically transformed into a stochastic program; Watson et al. (2012) suggest to use Progressive Hedging for finding solutions. The algebraic modelling system GAMS (Bussieck and Meeraus; 2004) provides support to express and solve stochastic programs as part of its extended mathematical programming (EMP) package. Similar to PySP, GAMS requires a deterministic base model and additional information about the stochastic structure of the problem, specified using EMP annotations. Hochreiter (2016) introduced a modelling language independent concept for modelling multi-stage stochastic programs that decouples the optimisation model from the stochastic "meta" model. Hochreiter (2016) argues that the variables and constraints in the deterministic base model can be split into three parts; the root,

```
1  include "stochastic.mzn";
2  int: n;
3  set of int: Items = 1..n;
4  int: capacity;
5
6  array[Items] of int: weights ::  stage(1);
7  array[Items] of int: profits ::  stage(2);
8
9  array[Items] of var 0..1:knapsack  ::  stage(1);
10 var int: obj  :: stage(2);
11 %capacity constraint
12 constraint sum (i in Items)(knapsack[i] * weights[i]) <= capacity;
13 %objective function
14 constraint obj = sum (i in Items) (knapsack[i] * profits[i]);
15
16 solve maximize obj ::  expected;
```

```
1  %determinisic data
2  n = 5;
3  capacity = 6;
```

```
1  %stochastic data
2  %the scenario set
3  set of int: SCENARIOS = 1..3;
4
5  profits = [|
6     2, 3, 6, 3, 1| %scenario 1
7     3, 4, 3, 7, 1| %scenario 2
8     4, 2, 8, 1, 3| %scenario 3
9     |];
10 %define the likelihood of each
11 %scenario
12 array[SCENARIOS] of int:
13    weights = [2,5,3]
14    ::  scenario_weights;
```

Figure 3.1: Knapsack problem with uncertain profits modelled in stochastic Minizinc

recourse and terminal stage. In contrast to the root and terminal stage that are unique for each stochastic program, multiple recourse stages are used to define a multi-stage problem. Finally, Stochastic MiniZinc (Rendl et al.; 2014) is an extension of the constraint modelling language MiniZinc (Nethercote et al.; 2007). In Stochastic MiniZinc, annotations are used to embed the uncertainty model into the standard (deterministic) constraint optimisation model, which in turn is automatically transformed into a stochastic program

Figure 3.1 displays how to model a knapsack problem with uncertain profits in Stochastic MiniZinc. Lines 2 to 16 formalise a standard (deterministic) knapsack problem. In addition, the parameters `weights` and `profits` and the variables `knapsack` and `obj` are annotated with `stage` to denote whether they belong to the first or second stage. The solve item is annotated with `expected` to indicate that we are interested in maximising the expected profit, essentially the probability weighted sum over all scenarios. An additional data file that describes the uncertainty using scenarios and their respective probabilities is required. The three files together are then transformed into a stochastic program; either the deterministic equivalent or into a form that can be solved using the scenario decomposition (the scenario decomposition in stochastic MiniZinc is not yet available).

We introduced the most relevant frameworks for modelling stochastic programs. All these frameworks are extensions of standard modelling languages and aim to exploit the strength of the underlying modelling system. The idea of formulating a deterministic decision model and using an additional "meta" model to describe the uncertainty is common throughout all these systems. In the following, we discuss modelling techniques that aim to improve the performance of solving a stochastic program.

Contributions

This chapter is composed of two main contributions, both study the role of adding extra constraints to the decision model, which in turn support the solver in finding solutions. The first contribution discusses how to use symmetry and dominance breaking constraints in stochastic programming modelling frameworks and the second contribution explores how to use redundant constraints, a technique that can be used to improve the scenario decomposition.

Constraint Programming models often contain symmetries and dominance relations that can be exploited to improve the search performance. We will first showcase challenges that arise with symmetry and dominance breaking constraints in the context of Stochastic MiniZinc. Thereafter we introduce a correct handling of these extra constraints, which is important to A) transform the model correctly into a stochastic program and B) utilise symmetry and dominance breaking constraints effectively during the solving process.

In the second part of the chapter we show how the scenario decomposition can be strengthened by adding auxiliary constraints to the scenario subproblems to reduce the number of infeasible candidates generated during the search. This is achieved by changing the scope of certain constraints from being local to a specific scenario to be globally visible and applied to each scenario.

3.2 Symmetry and Dominance

Many constraint problems exhibit *dominance relations* and give rise to *symmetric* solutions. The n-queens problem as introduced in Fig. 2.2 is a classic example that contains multiple symmetries one is displayed in Fig. 3.2. The challenge is to place a queen in each column while only assigning a single queen to each row and diagonal. The queens on the right side of Fig. 3.2 exactly mirror (with respect to the dashed line) the queens on the left. These two solutions are symmetric; it is possible to map one solution to another with equal quality. To avoid enumerating symmetric solutions and non-solutions, which improves the search performance, the constraint $q[1] < q[2]$ (and possibly others) can be added to the model definition, as in Fig. 2.2, without compromising the solution quality. *Note: a non-solution refers to a symmetric, yet infeasible variable assignment that has been proven to be infeasible in a previous search step.* This additional constraint is called *symmetry breaking constraint.*

Many optimisation problems, e.g. assignment problems such as packing a knapsack, exhibit dominance relations. Consider the task of choosing a subset of items from the set

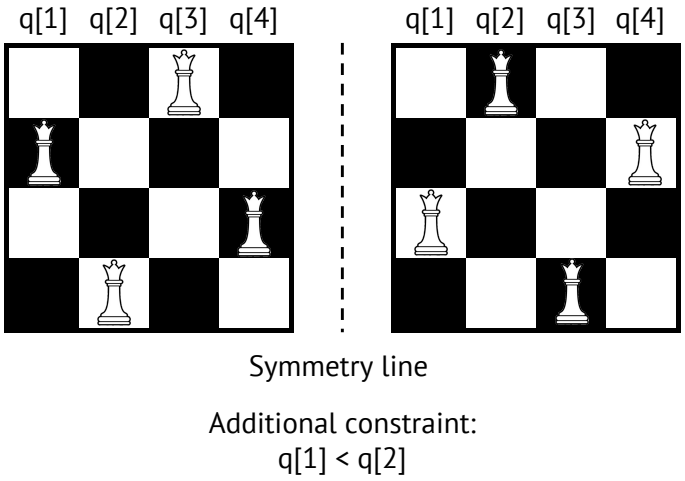


Figure 3.2: Symmetric solutions for the n-queens problem

of items as modelled in Fig. 3.1 and a specific instance displayed in Fig. 3.3. The goal is to maximise the profit with respect to a capacity constraint. It is implied that if two items have equal weight, e.g. items 1 and 2 (green) or items 4 and 5 (blue), the item with the higher profit must be chosen before considering the less valuable item. This implication can be added explicitly to the model as a *dominance breaking* constraint to avoid the exploration of solutions that are guaranteed to be inferior.

Symmetry and dominance relations can be exploited to achieve reductions in the search space and therefore improve search performance. Symmetry breaking is well studied, and its effects demonstrated, see Fahle et al. (2001); Chu and Stuckey (2016); Walsh (2006); Chu et al. (2014); Walsh (2010); Cohen et al. (2006). Similarly, dominance relations, a generalisation of symmetries, have been studied and shown to be effective in Prestwich and Beck (2004); Chu and Stuckey (2015); Proll and Smith (1998); Getoor et al. (1997).

Modelling Symmetry and Dominance relations

Chu and Stuckey (2016) argue that declaring symmetries explicitly should be a standard part of modelling, e.g. by using annotations or dedicated predicates. If the modeller is aware of symmetries, they should be declared such that the solver can take advantage of them. The same argument can be made for dominance constraints. A solver may choose to ignore symmetries, for example when constraint based local-search techniques are employed as in OSCAR.CBLS (Björdal et al.; 2015) to not interfere with the search strategy.

In the following, we will show that not declaring symmetries and dominance relations explicitly when using Stochastic MiniZinc can lead to wrong model definitions. First, we introduce an example to demonstrate the problems and thereafter, we will introduce how to deal with symmetry and dominance relations in Stochastic MiniZinc and emphasise the importance of declaring those extra constraints explicitly as introduced in (Chu and Stuckey; 2016).

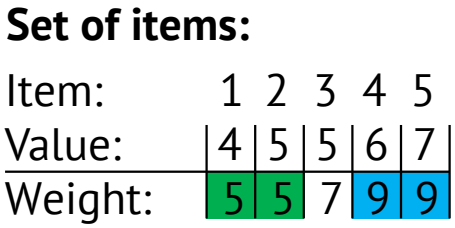


Figure 3.3: Dominance relations in the knapsack problem

3.2.1 Stochastic template design problem

As introduced earlier, stochastic programming modelling frameworks such as Stochastic MiniZinc are based on the idea of using a standard (deterministic) model to describe the decisions and augment it with some meta-model that describes the uncertainty. Constraint programming practitioners will therefore use symmetry and dominance breaking constraints to improve the decision model, as they normally would.

Fig. 3.4 contains the essential parts of the template design model introduced by Proll and Smith (1998) (the complete model can be found in the appendix) and Fig. 3.5 supports the explanation. An animal food factory would like to print cardboard packages for each of the different products they produce. For example, Fig. 3.5 displays on the left side a problem with four different food types, rabbit, bird, dog, and cat. For each product, we know the number of packages that must be produced, the demand. We are asked to design two templates, each with six slots. The templates are modelled as one-dimensional arrays where each position corresponds to a product and the respective value defines the number of slots designated to the product. The sum over each template array is equal to the number of slots on the template. The goal is to design the templates such that the total number of prints is minimised.

The constraints starting on line 45 in Fig. 3.4 are symmetry breaking constraints. These constraints are active if the demand of two consecutive products is the same, e.g. the demand of packages for rabbits and birds is both equal to 200. An example of two symmetric solutions is illustrated in Fig. 3.5 on the top right. To break the symmetries, we create an auxiliary array for each packaging type; the composition of the respective element in the first and second template. A lex_lesseq constraint is used to enforce a lexicographical relationship between the two resulting arrays. For example, the number of slots assigned to rabbits on the first and second template must be smaller or equal to the number of slots assigned for birds.

The constraints starting on line 54 are dominance breaking constraints. If the demand for a package type is smaller than its successor product, the overall production must also be smaller. For example, in Fig. 3.5 the number of packages produced for bird food must not exceed the dog food packaging production, as the demand for dog food packages is higher compared to bird food.

```
 9 include "globals.mzn";
10 include "stochastic.mzn";
11
12 int: S;                  % Number of slots per template.
13 int: t;                  % Number of templates.
14 int: n;                  % Number of variations.
15 % How much of each variation we must print?
16 array[1..n] of int: d ::  stage(2);

21 % # Slots allocated to variation i in template j
22 array[1..n,1..t] of var 0..S: p ::  stage(1);
23
24 % # Pressings of template j.
25 array[1..t] of var 1..lupper: R ::  stage(2);
26
27 % Sum of all Rj.
28 var llower..lupper: Production ::  stage(2);

33 % First, set up Production to be the sum of the Rj
34 constraint Production = sum(i in 1..t)(R[i]);
35
36 % The number of slots occupied in each template is S.
37 constraint forall(j in 1..t)(sum(i in 1..n)(p[i,j]) = S);
38
39 % Enough of each variation is printed.
40 constraint forall(i in 1..n)(sum(j in 1..t)(p[i,j]*R[j]) >= d[i]);
41
42 %% Symmetry breaking constraints
43 % Variations with the same demand are symmetric.
44 constraint symmetry_breaking_constraint(
45             forall(i in 1..n-1) (
46              d[i] == d[i+1] ->
47              lex_lesseq([p[i,  j] | j in 1..t],
48                         [p[i+1,j] | j in 1..t])
49             )
50            );
51
52 %% Dominance breaking constraints
53 constraint dominance_breaking_constraint(
54             forall(i in 1..n-1) (
55              d[i] < d[i+1] ->
56              sum (j in 1..t) (p[i,j]*R[j])
57              <= sum (j in 1..t) (p[i+1,j]*R[j])
58             )
59            );

71 % Minimize the production.
72 solve :: int_search(array1d(1..n*t,p) ++ R,
73                     input_order, indomain_min, complete)
74      minimize Production ::  expected;
```

Figure 3.4: Template design problem (extract)

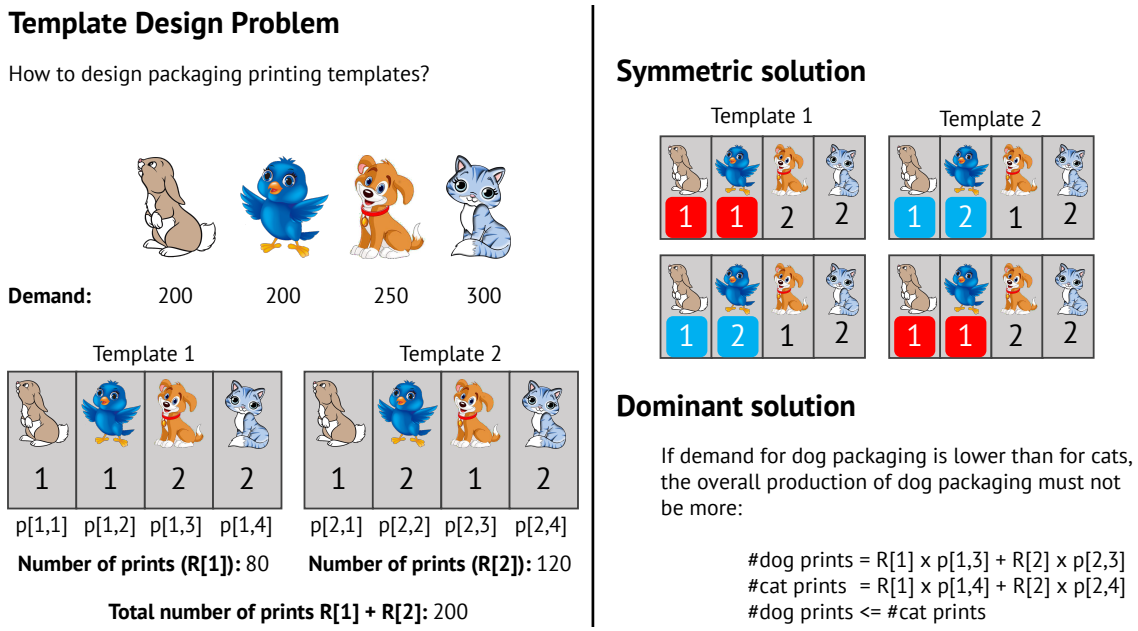


Figure 3.5: A template design problem

3.2.2 Stochastic MiniZinc and symmetries/dominance

The template design problem displayed in Fig. 3.4 is modelled using Stochastic MiniZinc syntax. In contrast to the problem introduced by Proll and Smith (1998), we assume that the demand is uncertain. In the first stage, we design the templates and in the second stage we decide how many times to print each template given the observed demand, specified using scenarios. The objective is to minimise the expected number of prints whilst satisfying the constraints in each scenario. A similar model was studied by Tarim and Miguel (2005), however the first stage in their model was composed of designing the templates and deciding on how many times to print each template, with the aim to reduce the expected sum of surplus and scrap in the second stage, given a random demand.

Great care has to be taken when compiling the core model, Fig. 3.4, into a stochastic program, as neither the scenario dependent symmetry nor dominance constraint in the template design model are valid in the stochastic model. In Fig. 3.6 we show the result, when compiling the extra constraints naively into the stochastic program. The problem is the following; a solution that is symmetric with respect to one scenario might not be symmetric with respect to another scenario. However, unless specified, the compiler cannot know that the extra constraints are not an integral part of the problem description. Therefore, it is important to mark symmetry and dominance constraints as auxiliary constraints.

```
41 % Symmetry breaking constraints
42 % naive compliation into a deterministic equivalent
43 % Variations with the same demand are symmetric.
44 constraint forall(s in SCENARIOS)(
45              forall(i in 1..n-1) (
46               %local (scenario specific) scope
47               d[s,i] == d[s,i+1] ->
48               %activates constraints in the shared scope
49               lex_lesseq([p[i,  j] | j in 1..t],
50               [p[i+1,j] | j in 1..t])
51              )
52             );
53
54 % Dominance breaking constraints
55 % naive compliation into a deterministic equivalent
56 constraint forall(s in SCENARIOS)(
57              forall(i in 1..n-1) (
58               %local (scenario specific) scope
59               d[s,i] < d[s,i+1]  ->
60               %activates constraints in the shared scope
61               sum (j in 1..t) (p[i,j]*R[j])
62               <= sum (j in 1..t) (p[i+1,j]*R[j])
63              )
64             );
```

Figure 3.6: A native transformation in stochastic MiniZinc

Compiling Symmetry and Dominance constraints

To compile symmetry and dominance constraints in the context of Stochastic MiniZinc correctly, the scope of their impact must be considered. In a two-stage stochastic program four situations regarding scope can occur, as illustrated in Fig. 3.7.

1. Local scope in first stage:
   - If a symmetry or dominance constraint reasons only over the first-stage variables and parameters, no attention in the model transformation is required. Only solutions that are symmetric across all scenarios are removed and the constraint is therefore globally valid.
2. Local scope second stage:
   - If a symmetry or dominance constraint only reasons over the second-stage variables and parameters, no attention in the model transformation is required. The scope of the constraints is scenario specific and therefore they do not invalidate globally valid solutions.
3. An implication from the first stage to the second stage:
   - No attention is required, as this situation complies with the nonanticipativity principle. In other words, the scope of the first stage is consistent over all scenarios and a first-stage event can imply a set of constraints in the second stage (local scope) without any special considerations.

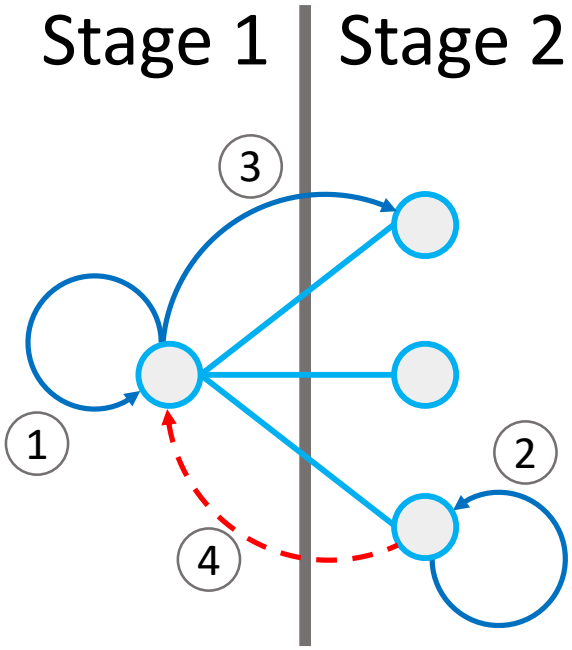


Figure 3.7: The impact of a symmetry or dominance constraint

4. An implication from the second stage to the first stage:

    - Attention is required! The symmetry and dominance constraints are not valid and must be omitted when compiling the core model intro a stochastic program. A solution that is symmetric in one scenario, may not be symmetric with respect to another scenario.

Marking symmetry and dominance constraints is important to support the compiler in creating strong and correct stochastic programs. The first three situations can directly be compiled into the stochastic program, however when the last situation arises, the symmetry and dominance breaking constraints must be ignored.

This section showed how symmetry and dominance constraints can be added to improve the decision model when formalising stochastic programs at a high-level of abstraction. In the following section we will see how general constraints can be lifted from the scenario level to the model level.

3.3 Global View on the Scenario Decomposition

The scenario decomposition is the foundation of numerous algorithms to solve stochastic problems. It has been shown that the performance of scenario decomposition based algorithms can be improved by grouping sets of scenarios into single subproblems, essentially creating a DE for each scenario group (Ryan, Ahmed, Dey and Rajan; 2016). The result is an interesting trade-off between A) an increased time to solve the subproblems yet B) decreased number of steps required to solve the stochastic problem. This trade-off is illustrated in Fig. 3.8. On the left side, the stochastic program is represented using only one

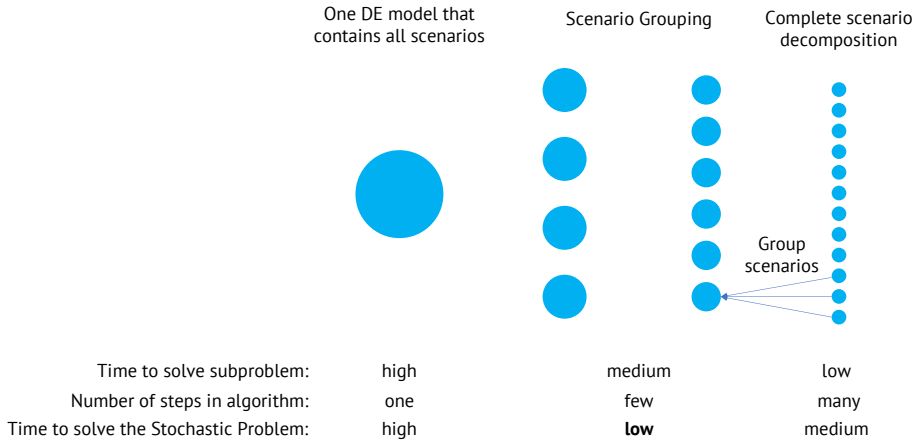


Figure 3.8: Scenario Grouping

model, the DE. On the right side, the problem is completely decomposed into single scenarios and in the middle, scenarios are grouped. The time to solve a subproblem decreases from left to right, however the number of steps required to solve the stochastic program, e.g. with E&C, increases along the same axis. Scenario grouping aims to exploit this trade-off and has been applied successfully. Crainic et al. (2014) use k-means clustering to group similar scenarios and dissimilar scenarios. Deng et al. (2017) use a mixed-integer program that groups scenarios optimally to solve chance-constrained programs and Ryan, Ahmed, Dey and Rajan (2016) use a mixed integer based approach to group scenarios optimally with respect to the lower bound obtained in the first iteration of E&C. And lastly, results that showcase the effectiveness of scenario grouping are presented in chapter 4 of this thesis.

In this section we propose a method that is orthogonal to scenario grouping and aims to reintroduce information back into subproblems lost when decomposing the stochastic program by scenarios. On the one end of the spectrum, the DE can be seen as the complete set of variables and constraints required to represent the stochastic program. On the other end, a single scenario can be viewed as the smallest unit of variables and constraints that is considered in light of the scenario decomposition. *Note: a single scenario can be further decomposed and solved using a decomposition algorithm, however we define a single scenario as the smallest entity.* In scenario grouping a set of scenarios is selected and all associated variables and constraints are used to formalise one model. In contrast, we propose to select a single constraint or set of constraints, instantiate it with the scenario specific parameters and introduce the resulting constraints into all other subproblems, as shown in Fig. 3.10. Let us use an example to illustrate the intuition behind the idea.

Consider a facility location problem with capacity constraints, where a number of facilities are used to satisfy customer demand in each scenario. In the first stage, we decide which facilities to open; each facility has limited capacity. In the second stage, the customers are assigned to exactly one facility; the availability of customers is random. We assume that all facilities have a capacity of 30 and all customers a demand of 10. The problem is composed of two scenarios, as displayed in Fig. 3.9, a high-demand scenario

with ten customers on the left side and a low-demand scenario with five customers on the right side. When using E&C to solve the problem, the first iteration might produce the following situation. The optimal solution to the high-demand scenario requires four open facilities (green circles) and the optimal solution to the low-demand scenario requires only two facilities to be opened. Clearly, all solutions with less than four open facilities are not feasible as they do not satisfy the customer demand in the high-demand scenario. Our goal is to completely avoid or reduce the number of candidates generated that are infeasible.

To avoid generating infeasible solutions, let us assume that we instantiate a constraint with parameters from the high-demand scenario and add the resulting concrete constraint to the low-demand scenario. We consider a feasibility constraint, which ensures that the number of facilities opened in the first stage are sufficient to satisfy the customer demand in the second stage. To make this more concrete: scenario one implicitly contains the constraint that four facilities must be opened. Adding this constraint explicitly to the low-demand scenario model avoids generating infeasible candidates with less than four available facilities. While this additional constraint does not provide any guarantee that no further infeasible candidates are generated, it reduces the number of infeasible candidates during the search and therefore improves the performance.

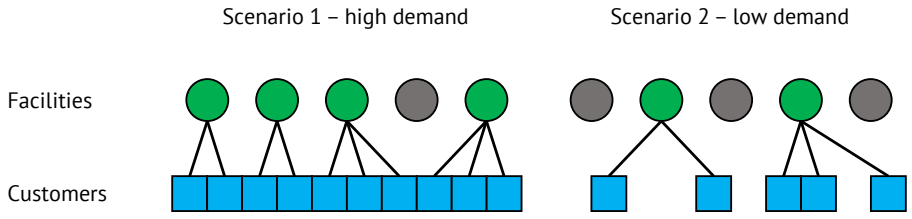


Figure 3.9: Facility location problem with two scenarios

Chapter 2, the background, introduced a two-stage stochastic program as $< V, [\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}k}] >$ where $\mathcal{P}_{\mathcal{D}i} = < V_i, D_i, C_i, f_i >$ describes a scenario. The variables V_i of each $\mathcal{P}_{\mathcal{D}i}$ are the union of the first- and second-stage variables and likewise, the constraints C_i are the union of the first- and second-stage constraints. Using the same notation, we formalised a deterministic equivalent, a single model containing all scenarios, as:

$$\mathcal{P}_{\mathcal{S}DE} =< \bigcup_{i \in K} V_i, \bigcup_{i \in K} D_i, \bigcup_{i \in K} C_i, \sum_{i \in K} f_i(x) >, \tag{3.1}$$

where K is the set of scenarios used to define the stochastic program.

As introduced above, scenario bundling yields an interesting trade-off between time to solve a subproblem and the number of iterations required to solve the stochastic program. A scenario bundle is essentially a DE composed of only a subset of scenarios. For example,

$$\mathcal{P}_{\mathcal{S}B_j} =< \bigcup_{i \in B_j} V_i, \bigcup_{i \in B_j} D_i, \bigcup_{i \in B_j} C_i, \sum_{i \in B_j} f_i(x) >, \tag{3.2}$$

where B_j is the set of scenarios that belong to bundle j.

In contrast to scenario bundling, we propose to select a constraint or set of constraints C_s, which are restricted to the subset of constraints that do not constrain any second-stage variables, and add it to all scenarios,

$$\begin{aligned} C_s &= \bigcup_{j \,:\, scope(c_j) \subseteq V} c_j \\ \mathcal{P}_{\mathcal{D}i} &= < V_i, D_i, C_i \cup C_s, f_i > \quad \forall j \in K \end{aligned} \tag{3.3}$$

The constraint set C_s is composed of one or multiple constraints that act only on the shared variables but contain second-stage parameters. In Fig. 3.10 we illustrate how the two-stage problem is decomposed into subproblems and a single scenario constraint lifted to be globally visible and enforced in all subproblems. A constraint in the set C_1 is selected and added to all other subproblems.

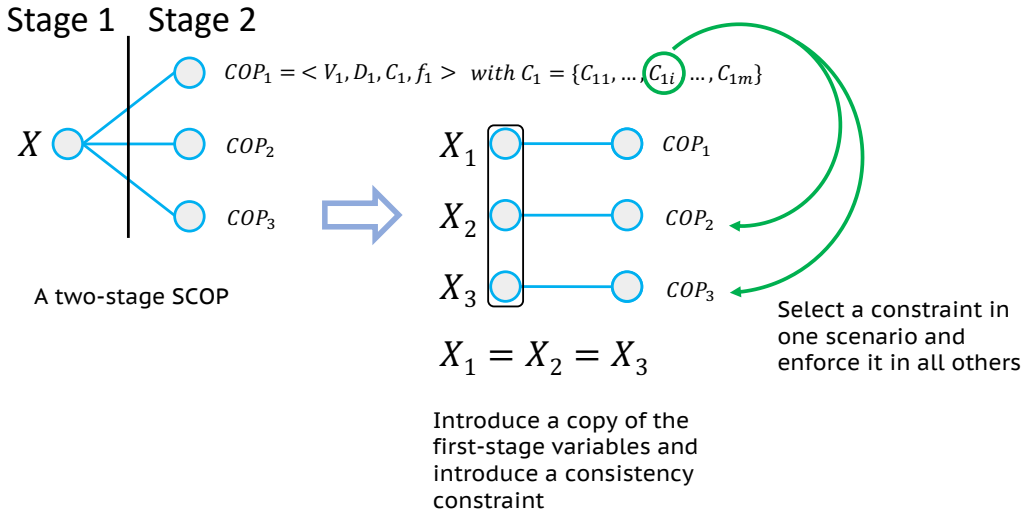


Figure 3.10: Enforce a scenario constraint in all subproblems

Numerous stochastic problems exhibit a structure where the scenario decomposition can be improved by making certain constraints globally visible. Consider for example the facility location problem with capacity constraints as introduced earlier. When decomposing the model, we lift the capacity constraints to become globally visible and therefore reduce the number of infeasible candidates. Many investment problems follow a similar structure, such as designing an optimal fleet of vehicles for a logistics company or investment decisions into the electricity network. In both cases, a certain demand revealed in the second stage must be met. Lastly, consider the template design problem as introduced earlier with four types of packages. Let us assume that there is a scenario in which the demand for a specific package type is zero. When solving this scenarios, no slot on the templates will be allocated for the package type with zero demand, as no reward is given. It is evident that such a candidate is infeasible when evaluating over a scenario that demands packages of the very same type. Lifting a constraint that enforces one or more slots

per package type would ensure feasibility without compromising global optimality. The impact of lifting a constraint is twofold. First, the number of candidates that are infeasible is reduced and secondly, the lower bound increases. This yields an overall reduced time to solve the stochastic program. The concept of lifting constraints to avoid enumerating infeasible constraints can be generalised to multistage problems, with the difference that the feasibility constraints added in stage two and subsequent are restricted in scope. In the subsequent section we provide evidence that lifting constraints can be beneficial.

3.3.1 Experiments

Lifting constraints to be globally visible when using the scenario decomposition helps to improve the performance when using algorithms such as E&C. To showcase this effect, we use a four-stage facility location problem with capacity constraints. In the first stage, a number of facilities is opened to serve customer demand revealed in the second stage. Thereafter, in the second stage, all customers are assigned to the facilities opened in the first stage and additionally a new set of facilities might be opened. The same decisions are repeated in the third stage before reaching the final (fourth) stage, in which no further facilities are opened. The complete model can be found in the appendix and a mathematical description of the problem is presented in chapter 5.

3.3.2 Results

The facility location problem was solved using a multi-stage variant of E&C, which is introduced in chapter 5. The same chapter contains a section that discusses the results comprehensively, however for now we focus only on the impact of lifting capacity constraints. Two sets of results are compared, firstly the stochastic problem was decomposed and solved without lifting any constraints to be globally visible. Thereafter, using the same algorithm, we solved the problem with lifted capacity constraints.

When using the improved scenario decomposition, with globally visible capacity constraints, all candidates were feasible. In contrast, we display the number of infeasible candidates when using the standard decomposition in Fig. 3.11. More candidates are infeasible with an increased number of scenarios used to model the uncertainty in the stochastic program. This does not come at a surprise as the overall number of candidates generated while solving the problem is higher when more scenarios are used. The overall effect on the runtime is shown in Fig. 3.12. The time to solve the problem is on average 20.9% lower when using the strengthened scenarios, independently of the number of scenarios used in the stochastic program.

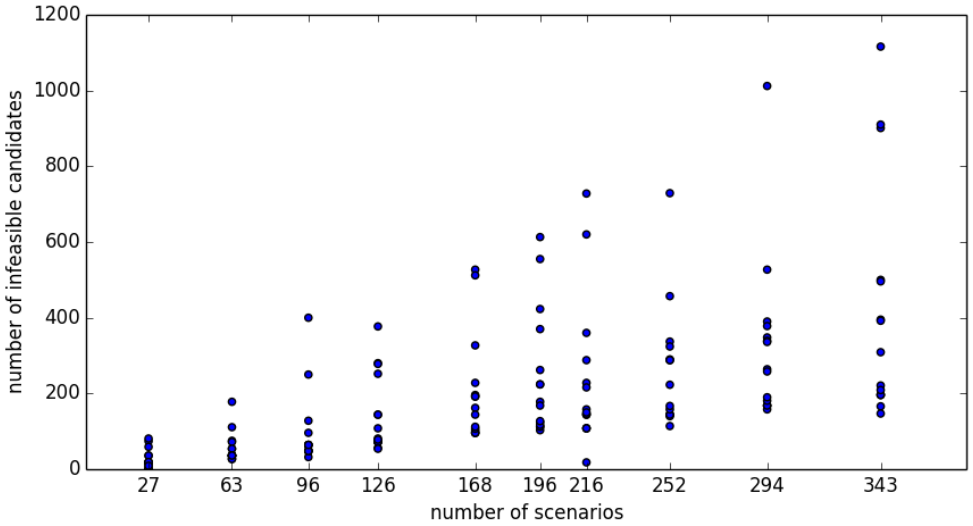


Figure 3.11: Number of infeasible candidates

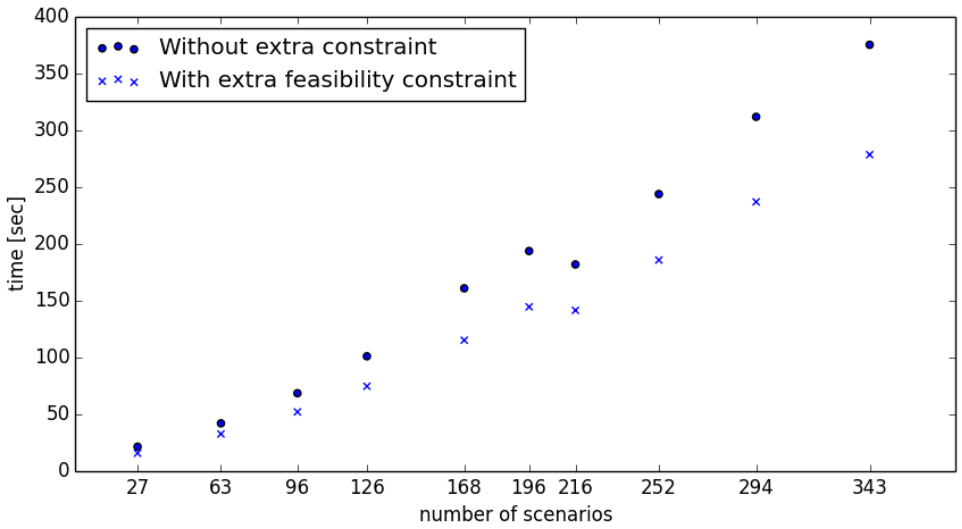


Figure 3.12: Standard versus strengthened scenario subproblems

3.4 Related Work

To the best of our knowledge, there are no modelling frameworks for stochastic programs that support symmetry and dominance breaking constraints nor the automatic or semi-automatic lifting of scenario specific constraints. However, conversations with other researchers have led to the conclusion that modelling techniques as we just proposed are implicitly used in practice.

3.5 Conclusion

We presented two techniques for improving stochastic programs modelled in a high-level modelling language for decision problems under uncertainty. Firstly, we have shown that symmetry and dominance breaking constraints must be identified and annotated correctly, to ensure the correct compilation of the decision and uncertainty model into a stochastic program. Secondly, we presented a technique that is orthogonal to scenario grouping and aims to increase the performance of scenario decomposition-based algorithms, such as E&C. We have demonstrated how lifting constraints improves the scenario decomposition algorithm E&C.

3.5.1 Limitations

As introduced earlier in this chapter, various works have shown that symmetry and dominance breaking constraints are beneficial when applied to deterministic combinatorial optimisation problems. Similarly, exploiting symmetries and dominance relations in stochastic programming can only be done if a model exhibits these relations. Furthermore, the effect on performance is strongly dependent on the individual use case. However, we have shown how to correctly deal with symmetry and dominance breaking constraints and the result of failing to mark them as such.

We proposed to lift constraints to improve the performance of scenario decomposition based approaches such as E&C for certain types of problems, such as the facility location problem with capacity constraints. Needless to say, lifting constraints will only be beneficial when solving model where A) some scenarios yield infeasible candidates and B) a constraint can be identified that reduces the number of infeasible candidates when lifted and introduced into all subproblems.

3.5.2 Future Work

Both contributions presented in this chapter are not yet made available in any modelling framework. Chu and Stuckey (2016) showed how to annotate symmetry and dominance breaking constraints when modelling deterministic optimisation problems in MiniZinc. The same annotations can be used in Stochastic MiniZinc; however, the extra compiler functionality is required.

We presented experiments that show how lifting constraints can improve the performance of algorithms such as E&C. However, further experiments are required to establish

a better understanding of situations where the lifting technique is beneficial. Furthermore, an interesting question that arises is whether it is possible to automatically identify constraints that can be lifted. For example, the capacity constraint in the facility location example might not be explicitly stated but inferred from other constraints. Ideally, the compiler would automatically identify and propose constraints that can be lifted.

Chapter 4

Evaluate and Cut with Diving

4.1 Introduction

Solving stochastic problems is challenging and numerous algorithms that exploit the problem structure of stochastic programs have been developed in the past. Chapter 2 introduced the scenario decomposition and the Evaluate and Cut (E&C) algorithm (Ahmed; 2013) as a method that is based on the scenario decomposition. E&C is an iterative three step procedure; first, each scenario is solved to obtain a lower bound (given a minimisation problem) and a set of first-stage variable assignments, also called candidates. Secondly, each candidate is evaluated over the complete set of scenarios to obtain an upper bound. And lastly, the evaluated candidates are excluded from the search using nogood constraints. By iteratively repeating these three steps, the optimal solution is found and optimality proven. As mentioned in chapter 2, E&C has been applied successfully to solve stochastic programs and is especially strong at finding the optimal solution quickly, often in the first few iterations. However, proving optimality can take a very long time, as the candidate nogoods exclude only small parts of the search space.

Contributions

The focus of this chapter is to improve the performance of E&C. The chapter presents two main technical contributions, illustrates the effect of scenario bundling, an established technique, and introduces a new stochastic programming benchmark. We are the first to use E&C, intended to solve problems with binary variables, in a CP setting. First, we propose to use a *Lazy Clause Generation* CP solver for solving the subproblems and show how *inter-instance learning*, can be used to improve the solver performance when finding candidates, the first step of E&C. The second contribution introduces *diving*, a technique that aims to reduce the time required to prove optimality by adding strong nogood constraints that prune larger parts of the search space than the original algorithm does. The benchmark used for the experiments can be found in the appendix and has been made public on the CSPLib (Gent and Walsh; 1999). The content of this chapter has been published by Hemmi et al. (2017) at the International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research 2017.

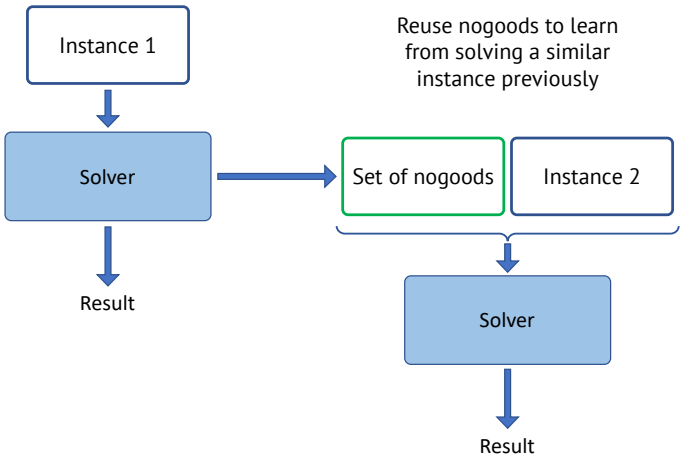


Figure 4.1: Inter-instance learning using a lazy clause generation CP solver

4.2 Scenario decomposition and Lazy Clause Generation

Traditional CP solvers use a combination of propagation and search. Propagators reduce the variable domains until no further reduction is possible or a constraint is violated. Backtracking search, usually based on variable and value selection heuristics, explores the options left after propagation has finished. In contrast to traditional CP solvers, Lazy Clause Generation (LCG) (Ohrimenko et al.; 2009) solvers *learn* during the search. Every time a constraint is violated the LCG solver analyses the cause of failure and adds a constraint to the model that prevents the same failure from happening during the rest of the search. The added constraints are called *nogoods*, and they can be seen as the CP equivalent of cutting planes in integer linear programming – they narrow the feasible search space.

Chu and Stuckey (2012) showed how nogoods can be reused across multiple instances of the same model if the instances are structurally similar. They called this technique *inter-instance learning*; its concept is illustrated in Fig. 4.1. Solving the first instance yields a result, the solution to the problem, and a set of nogood constraints. Instead of discarding the nogood constraints learned during the search, Chu and Stuckey (2012) propose to augment the second instance with the just learnt set of nogood constraints. Reusing the nogoods learned while solving one instance can yield a substantial reduction in the search space of another instance. Empirical results published in Chu and Stuckey (2012) indicate that for certain problem classes, a high similarity between instances yields a high reusability of nogoods and therefore increased performance.

In E&C, very similar instances are solved repeatedly. As shown in Fig. 4.2, in every iteration, all scenarios are solved over again. The only difference between two iterations is the additional candidate nogoods that result from the E&C coordination algorithm. As a consequence, inter-instance learning can be used to increase the performance when solving the scenarios to obtain candidates. We call this concept *vertical learning*, as the learnt nogoods are reused within the same subproblem. No changes are required to the standard E&C algorithm, except we assume that solving the subproblems works incrementally and

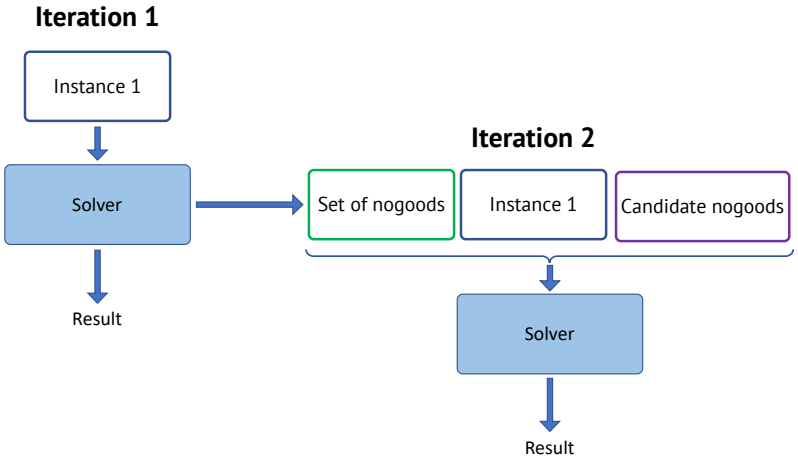


Figure 4.2: Inter-instance learning applied to E&C

the solver remembers the nogoods learned in the previous iteration for each $\mathcal{P}_{\mathcal{D}i}$. To the best of our knowledge, we are the first to make use of inter-instance learning in a vertical fashion. Inter-instance learning will be evaluated empirically in Section 4.4.

4.3 Search Over Partial Assignments

As introduced earlier, the E&C algorithm quickly finds high quality solutions by evaluating candidates, obtained from solving scenario COPs to optimality, over the complete set of scenarios. However, to prove optimality, EVALUATEANDCUT relies on the lower bound, which is computed as the probability weighted sum over all scenario objectives. The quality of the lower bound, and the number of iterations required to reach the upper bound and thus prove optimality, is solely dependent on the candidate nogoods added in each iteration.

Candidate nogoods as used in E&C are rather weak: They only cut off a single, complete first-stage assignment. Furthermore, in the absence of Lagrangian multipliers or similar methods, candidate nogoods are the *only* information about the lower bound of $\mathcal{P}_{\mathcal{S}}$ that is available to each scenario problem $\mathcal{P}_{\mathcal{D}i}$.

Stronger nogoods: To illustrate the candidate nogoods produced by E&C, consider a set of shared variables V of size five, and the candidate $x = [3,6,1,8,3]$. The resulting candidate nogood added to the constraint set of each scenario subproblem $\mathcal{P}_{\mathcal{D}i}$ is:

$$P_i = P_i[C \cup = \{x_1 \neq 3 \vee x_2 \neq 6 \vee x_3 \neq 1 \vee x_4 \neq 8 \vee x_5 \neq 3\}]$$

The added constraint cuts off exactly one first-stage assignment. A nogood composed of only a *subset* of the shared variables would be much stronger. For example, assume that it would be possible to prove that even the *partial* assignment $x_1 = 3 \wedge x_2 = 6 \wedge x_3 = 1$ cannot be completed to an optimal solution to the stochastic problem. We could add the

Algorithm 2 Evaluate and Cut for Two-Stage Problems

```
1:  procedure SOLVESCOP(P_S)
2:      Initialize: UB = ∞, LB = -∞, sol = null
3:      [P_D1,...,P_Dk] = GET_SCENARIOS(P_S)
4:      while LB < UB do
5:          LB = 0, S= ∅
6:          % Find first-stage candidates (lower bound)
7:          for i in 1..k do
8:              < σ,obj> = SOLVE(P_Di)
9:              LB += obj
10:             S ∪= {σ|_V}
11:         % Evaluate first-stage candidates (upper bound)
12:         for σ_V ∈ S do
13:             t_UB = 0
14:             for i in 1..k do
15:                 < _,obj> = SOLVE(P_Di[C ∪= {σ_V}])
16:                 t_UB += obj
17:                 ADDNOGOOD(P_Dk,σ_V)
18:             if t_ub < UB then
19:                 sol = σ_V
20:                 UB = t_UB
21:         % Evaluate partial first stage assignments
22:         DIVE(P_S,UB,S,sol)
23:     return sol
```

following nogood,

$$P_i = P_i[C \cup = \{x_1 \neq 3 \vee x_2 \neq 6 \vee x_3 \neq 1\}],$$

which in contrast to the original candidate nogood that eliminates exactly one solution, is able to prune a much larger part of the search space. We call this stronger kind of nogood a **partial candidate nogood**.

Diving - algorithm:

We now develop a method for finding partial candidate nogoods that we call *diving*. The main idea is to iteratively assign values to a subset of first-stage variables across all scenarios, essentially enforcing partial consistency, solve the subproblems and continue with assigning further variables until the resulting bound exceeds the incumbent upper bound of $\mathcal{P}_\mathcal{S}$. In that case, the fixed variables can be added as a partial nogood, since no complete variable assignment can be found that is better than the incumbent.

The modification to standard E&C as displayed in Algorithm 2 (same as Algorithm 1 in chapter 2) consists of a single call, to a procedure called DIVE in line 22, which is executed in each iteration after evaluating the candidates. The definition of DIVE is described in Algorithm 3. A constraint c is created that fixes a subset of the first-stage variables, line 5. The selection of the respective variable assignments is done according to a heuristic. The loop in lines 7 to 10 is very similar to the computation of candidates, except that the

partial consistency constraint c is enforced in all scenarios. As in standard E&C, a bound is calculated based on the probability weighted sum of all scenarios. If this bound meets or exceeds the incumbent upper bound (line 20), the partial consistency constraint c is turned into a nogood and added to all subproblems.

Enforcing partial consistency can result in one of three states:

1. The new bound does not exceed the incumbent upper bound, but all scenarios agree on a common first-stage variable assignment (even if c does not constrain all first-stage variables). This means that a new incumbent solution is found (lines 12–18) and the constraint c can be added as a partial candidate nogood.

2. The new bound meets or exceeds the upper bound. In this case, the partial consistency constraint c cannot be extended to any global solution that is better than the incumbent (lines 20–23). We can therefore add c as a partial candidate nogood.

3. The new bound is below the upper bound, and the shared variables of all scenarios did not converge. In this case, an additional constraint is added to the partial consistency constraint c.

Algorithm 3 Searching for partial candidate nogoods using diving

```
1:  procedure DIVE(P_S,UB,S,sol)
2:      t_LB = -∞
3:      while t_LB < UB do
4:          t_LB = 0
5:          c = SELECTFIXED(S,V) % Select first stage variables to fix
6:          S= ∅
7:          for i in 1..k do
8:              <σ,obj> = SOLVE(P_Di[C ∪= {c}])
9:              t_LB += obj
10:             S ∪= σ|_V
11:         % new incumbent found
12:         if t_LB < UB ∧ S = {σ_V} then
13:             UB = t_LB
14:             sol = σ_V
15:             % Add partial candidate nogood
16:             for i in 1..k do
17:                 P_Di = P_Di[C ∪= {¬c}]
18:             return
19:         % Proof: incumbent solution (UB) is better than all possible candidates (t_LB)
20:         if t_LB >= UB then
21:             for i in 1..k do
22:                 P_Di = P_Di[C ∪= {¬c}]
23:             return
```

The DIVE procedure terminates because in each iteration, SELECTFIXED fixes at least one additional first-stage variable, which implies that either case (1) or (2) above must eventually hold.

Note, the candidates obtained during each iteration of diving are not evaluated over all scenarios, as one might expect based on standard E&C. Firstly, evaluating all candidates can be computationally expensive. Furthermore, the optimal solution would either way be found in a subsequent iteration given it is an extension of the partial consistency.

Diving heuristic:

Let us now propose a heuristic for choosing the consistency constraints that are added in each iteration of a dive. Our goal is to produce short, yet relevant partial candidate nogoods and achieve convergence across all scenarios quickly. The motivation behind the heuristic we are proposing is the following; a partial nogood that invalidates the variable assignment that is most commonly found across the set of candidates prunes strongly. The motivation is based on the assumption that candidates yield likely high quality solutions, without having any formal explanation.

The heuristic we are proposing selects candidate variables that are already converged, plus an additional assignment. The procedure SELECTANDFIX in Algorithm 4 implements this heuristic. At first, all candidate variables that are already converged are chosen, including those fixed in previous steps of this dive (line 5 right side of $\wedge$). In addition, another first-stage variable assignment is selected (line 5 left side of $\wedge$), based on the variable/value combination that occurs most often over all scenarios.

Given the current set of candidates S, the algorithm constructs a mapping $Count$ from variables to multisets of their assignments (line 3). Thereafter, the most frequently observed, yet not completely converged, variable/value combination is selected (line 4). For example, if variable x_3 is assigned to the value 4 in three candidates, and twice to the value 7, then $Count$ would contain the pair $< x_3, \{4,4,4,7,7\} >$. The value 4 would be assigned to x_3 as it is the most prevalent variable/value combination. Finally, the algorithm constructs a constraint that assigns x_e to v_e, in addition to assigning all variables that are already converged (line 5).

Algorithm 4 Diving heuristic

1: **procedure** SELECTANDFIXED(S,V)
2: $Vals =< \{\sigma_i(x) : \sigma_i \in S\} : x \in V >$
3: $Count =< card(\{i : \sigma_i(x) = val, \sigma_i \in S\}) :< x \in V, val \in Vals_x >>$
4: $< x_e, v_e > = \underset{\substack{<x,v> \\ val \in Vals_x \\ Count_{<x,v>} < k}}{\arg\max} \; Count_{<x,val>}$
5: $c = (x_e = v_e \wedge \bigwedge_{\substack{x \in V \\ val \in Vals_x \\ Cout_{<x,v>} = k}} x = v)$
6: **return** c

Scenario Bundling:

The final extension to E&C is scenario bundling. When using the scenario decomposition, the subproblems can be a group of scenarios that are modelled as a DE, as described in chapter 3. Since the evaluation procedure in EVALUATEANDCUT has $O(k^2)$ behaviour for k scenarios, bundling can have a positive effect on the runtime, as long as the time to solve a bundle is not significantly higher than that for an individual scenario. Furthermore, the bundling of scenarios yields stronger lower bounds, as each resulting candidate is optimal with respect to a set of scenarios rather than just a single one. Scenario bundling therefore combines the fast convergence of EVALUATEANDCUT for large numbers of scenarios with the good performance of methods such as the DE on low numbers of scenarios. Scenario bundling has been applied to progressive hedging (Crainic et al.; 2014) and to EVALUATEANDCUT in (Ryan, Ahmed, Dey and Rajan; 2016).

4.4 Experiments

This section reports on our empirical evaluation of the algorithms discussed above. As benchmark we use a stochastic assignment problem with recourse, similar to the stochastic generalised assignment problem (SGAP) described in Albareda-Sambola et al. (2006); the most relevant parts of the model are displayed in Fig. 4.3 and the complete model can be found in the appendix, Fig. A.1.4; the subsequent line references refer to these models, the lines are consistent. A set of jobs (line 18), each composed of multiple tasks (line 22), is to be scheduled on a set of machines (line 17). Precedence constraints, starting in line 93, ensure that the tasks in a job are executed sequentially. Furthermore, tasks may be restricted to a sub-set of machines, the constrains start on line 131. The processing time of the tasks varies across the set of machines, and in the stochastic version of the problem, this is a random variable (line 102). In the first stage, tasks must be assigned to machines. An optimal schedule with respect to the random variables is created in the second stage. The objective is to find a task to machine assignment that minimises the expected makespan over all scenarios.

Work on the SGAP with uncertainty on whether a job must be executed is described by Albareda-Sambola et al. (2006). However, to the best of our knowledge, there are no public benchmarks for the SGAP with uncertain processing times. We created benchmarks for our experiments on the basis of the deterministic flexible job shop instances introduced by Brandimarte (1993). The scenarios are created by multiplying the base task durations by a number drawn from a uniform distribution with mean 1.5, variance 0.3, a lower limit of 0.9 and upper limit of 2.

To model the benchmarks, we used MiniZinc. Each scenario is described in a separate MiniZinc data file and compiled separately. This enables the MiniZinc compiler to optimise the COPs individually before solving. The models use a fixed search strategy (preliminary studies using activity-based search, usually a high performing search strategy, did not improve the performance). The solver is implemented using Chuffed (Chu; 2011) and Python 2.7. The scenarios are solved using Chuffed and learned nogoods are kept for

subsequent iterations to implement vertical learning. A Python program implements the main E&C algorithm with diving as presented in section 4.3. Up to twenty scenarios are solved in parallel. The experiments were carried out on a 2.9 GHz Intel Core i5, Desktop with 8 GB RAM running OSX 10.12.1. A time-out of 3600 seconds was enforced.

4.4.1 Results

Table 4.2 contains the results for 9 representative problem instances with a range of 20 to 400 scenarios. In every instance, E&C is able to retrieve the optimal solution within the first few iterations and the remaining time is used to prove optimality. No results for the deterministic equivalent are presented as the runtime is not competitive once the number of scenarios exceeds 20.

The impact of diving:

The first two rows per instance in Table 4.2 contain the time it takes to solve the problem instances without scenario bundling. No substantial time difference can be reported for finding the optimal solution when dives are enabled; we therefore do not show these times. However, using dives improves the overall search performance in every instance. Figure 4.4 contains a set of plots that display the impact of dives when solving 100 scenarios without bundling. The monotonically increasing graphs are the lower bounds. The horizontal black line is the value of the optimal solution. The upper bound progression towards the optimal solution is not displayed, as it is consistently found within a few seconds. Once the lower bound meets the upper bound, optimality is proven, and the search terminates. The lower bound can increase erratically, especially when diving is enabled. This is the reason that the lower bound exceeds the value of the optimal solution, once optimality is proven (not an error in the plot). In both cases, the lower bound increases quickly at the beginning of the search. However, over time the standard E&C method without diving flattens and the lower bound converges slowly towards the upper bound. This is strongly contrasted by the lower bound characteristics when using diving. At first during the initial iterations, the partial candidate nogoods are not showing any effects and the two curves are similar. However, after the initial phase, strong partial candidate nogoods are generated, and thus pay off by dramatically reducing the optimality gap.

Using DE to prove optimality:

CP solvers perform well when strong bounds are provided. The third row displays the time it takes to prove optimality given the DE and a constraint that forces the objective value to be lower than the optimal objective value.

Scenario bundling:

The last two rows per instance display the time it takes to solve the instances using scenario bundling. Four scenarios are randomly grouped to form a DE. Each scenario group becomes a subproblem solved with evaluateAndCut with and without diving

```
 9          % Including files
10          include "globals.mzn";
11          % Parameters
12          int: no_mach;    % Number of machines
13          int: no_jobs;    % Number of jobs
14          int: no_task;    % Number of total tasks
15          int: no_optt;    % Number of total optional tasks
16
17          set of int: Mach  = 1..no_mach;
18          set of int: Jobs  = 1..no_jobs;
19          set of int: Tasks = 1..no_task;
20          set of int: OptTs = 1..no_optt;
21
22          array [Jobs] of set of int: tasks;
23          array [Tasks] of set of int: optts;
24
25          array [OptTs] of int: optt_mach;
26          array [SCENARIOS1,OptTs] of int: optt_dur;

93          % Precedence relations
94          %
95          constraint
96             forall(s in SCENARIOS)(
97                forall(j in Jobs, i in tasks[j] where i < last_task[j])(
98                   start[s,i] + dur[s,i] <= start[s,i + 1]
99                )
100            );

102         % Duration constraints
103         %
104         constraint
105            forall(o in OptTs,s in SCENARIOS)(
106               let { int: t = optt_task[o] } in (
107                  if card(optts[t]) = 1 then
108                     b[o] = true
109                  else
110                     b[o] -> dur[s,t] = optt_dur[s,o]
111                  endif
112               )
113            );

131         % Resource constraints
132         %
133         constraint
134            forall(m in Mach,s in SCENARIOS)(
135               let {
136                  set of int: MTasks = { o | o in OptTs where optt_mach[o] = m }
137               } in (
138                  cumulative(
139                     [ start[s,optt_task[o]] | o in MTasks ],
140                     [ optt_dur[s,o]         | o in MTasks ],
141                     [ bool2int(b[o])        | o in MTasks ],1)));
142
143         constraint forall(s in SCENARIOS)(
144            forall(j in Jobs)(start[s,last_task[j]] + dur[s,last_task[j]]
145               <= de_objective[s]));
```

Figure 4.3: Generalised assignment problem

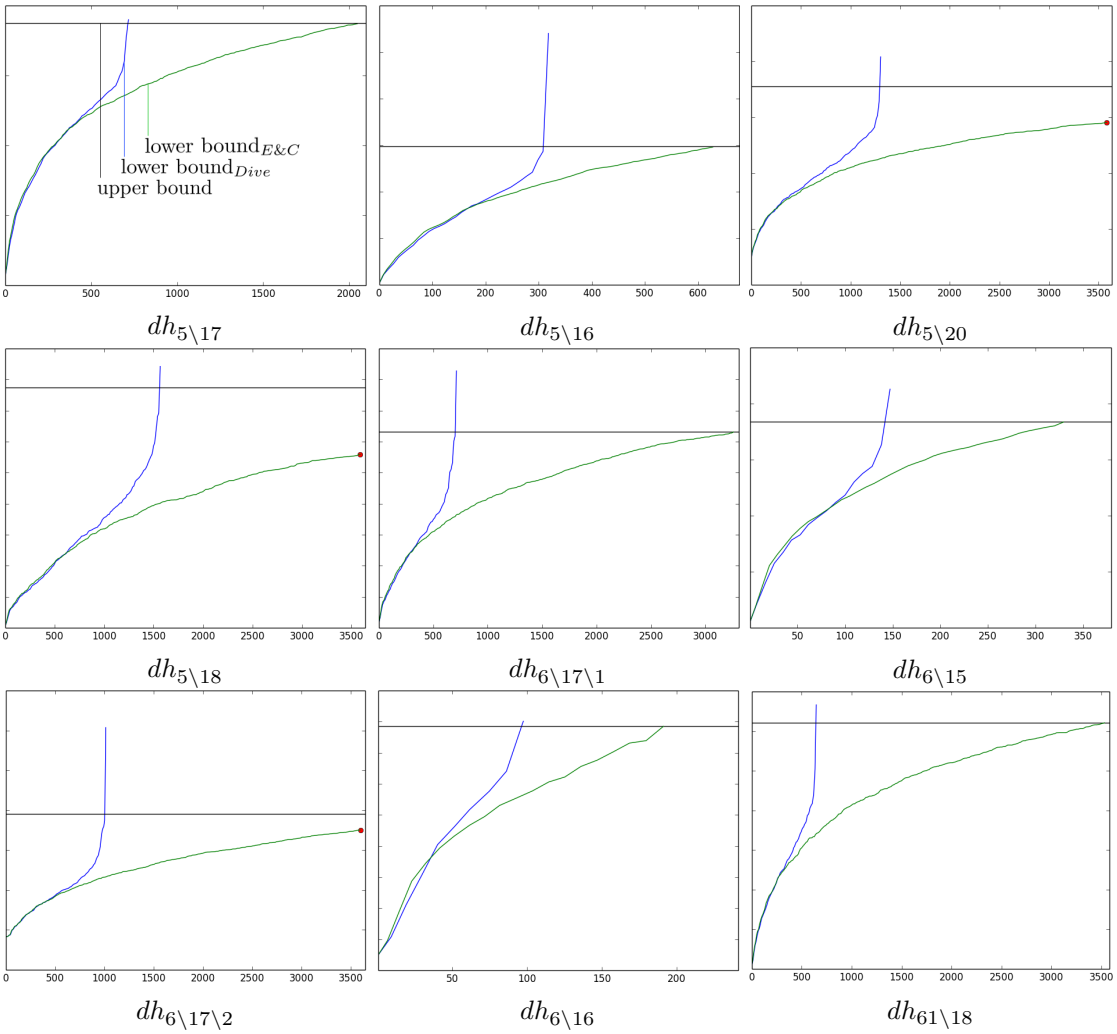


Figure 4.4: Time to prove optimality: E&C vs. diving

		20	20	40	40	80	80	100	100	200	300	400
E&C	mean	23.4	29.6	15.8	19.2	9.7	27.4	7.2	26.0	21.8	19.6	17.6
	variance	6.0	17.2	5.0	37.1	2.4	13.8	3.3	12.0	10.7	10.6	8.4
Dive	mean	21.4	32.7	13.7	28.7	8.1	29.9	5.9	23.8	17.1	14.7	12.4
	variance	6.3	23.6	4.0	21.9	2.4	16.6	2.1	11.5	8.9	6.8	5.8

Speedup: 100 - 100 / time no learning * time with learning

Colored: Speedup in [%] when using vertical learning without bundling scenarios

White: Speedup in [%] when using vertical learning and bundling scenarios

Table 4.1: Speedup using vertical learning

enabled. The runtime decreases in every instance. For the benchmark instances, diving is less powerful when using scenario bundles. This can be explained by the decreasing number of iterations required to find the optimal solution. More time is spent solving the subproblems and less effort is required to coordinate the scenarios.

Vertical learning:

Table 4.1 shows the speed-up when using vertical learning. Each column displays the speed-up over a scenario group. Overall the solving time decreased by 19.3% with 10.7% variance; the maximal increase observed was 72%.

4.5 Related Work

Ryan, Rajan and Ahmed (2016) propose a series of algorithmic improvements, such as using the LP relaxation when evaluating candidates and only solving the non-relaxed subproblem if required; many candidates can be discarded on the basis of solving the LP relaxation. Similar to their work, the LP relaxation can also be applied when using diving to find partial nogoods. Furthermore, Ryan, Rajan and Ahmed (2016) introduce an asynchronous parallelisation scheme that doesn't require each iteration to be finished before proceeding to the next iteration. Basçiftci et al. (2017) solve a generator maintenance and operations scheduling problem under uncertain failure times. They improve on E&C by introducing strong, problem specific nogood cuts. And lastly, Ryan, Ahmed, Dey and Rajan (2016) use various strategies to group scenarios to improve the search performance, similarly to the scenario grouping method we used in our experiments, however with a focus on grouping heuristics.

4.6 Conclusion

This chapter introduced the first successful application of E&C in a CP setting and two algorithmic innovations to improve the performance of the standard E&C algorithm. First, we presented how vertical learning, an application of inter-instance learning, can be used to speed up the search in each subproblem. Secondly, and perhaps most significantly, we

presented a method to decrease the time to prove optimality using diving to create strong nogoods. Finally, we empirically evaluated both improvements on a two-stage stochastic generalised assignment problem, for which we created instances that we published on the CSPLib (Gent and Walsh; 1999) with the problem id *prob077*.

4.6.1 Limitations

Our experiments confirm that using partial nogoods can substantially reduce the time to prove optimality when solving a stochastic problem using E&C. After demonstrating the effectiveness of diving, we would like to address its limitations. Ahmed (2013) used two benchmarks in his seminal paper to showcase the performance of E&C, a stochastic knapsack problem as well as a stochastic server location problem. Ahmed (2013) reports the number of iterations required to solve the problems. Most instances were solved in less than 10 iterations and of those, most required only two iterations. Clearly, if an instance is solved within only few iterations, diving will not improve the search performance. There is an overhead associated with diving, each dive requires multiple iterations, and for this overhead to pay off a large number of standard E&C iterations are required.

4.6.2 Future Work

Multiple future research directions in relation to diving can be identified. First, the heuristic used to determine the variables to be fixed strongly impacts the performance of the search. A good heuristic is able to produce strong, relevant nogoods. A heuristic based on variable activity during the search in COPs might produce strong results.

For the vertical learning experiments, we have used the standard system to manage nogoods within Chuffed. Further investigating the role of nogoods might help to improve the performance of inter-instance learning.

Instance	Algorithm	20	40	80	100	200	300	400
dh_5_17	E&C	24	343	1748	2060			
	Dive	12	152	584	716			
	$DE_{upperBound}$	7	215	1342	1923	-	-	-
	$E\&C_{bundle}$	1	**8**	**19**	17	27	49	**46**
	$Dive_{bundle}$	1	11	20	17	**26**	**45**	64
dh_5_16	E&C	36	111	436	638			
	Dive	23	56	212	318			
	$DE_{upperBound}$	20	131	877	1136	-	-	-
	$E\&C_{bundle}$	**2**	**5**	**9**	**11**	**34**	**69**	**91**
	$Dive_{bundle}$	3	7	12	13	41	72	93
dh_5_20	E&C	338	196	-	-			
	Dive	87	880	713	1304			
	$DE_{upperBound}$	43	293	2355	-	-	-	-
	$E\&C_{bundle}$	**3**	**17**	**36**	72	240	423	645
	$Dive_{bundle}$	7	18	43	**71**	**190**	**328**	**510**
dh_5_18	E&C	163	719	-	-			
	Dive	50	179	984	1566			
	$DE_{upperBound}$	**3**	50	1547	3467	-	-	-
	$E\&C_{bundle}$	7	**31**	162	226	407	712	1233
	$Dive_{bundle}$	8	32	**157**	**218**	**401**	**629**	**873**
dh_6_17_1	E&C	94	735	2328	3242			
	Dive	33	152	447	712			
	$DE_{upperBound}$	6	62	647	1073	-	-	-
	$E\&C_{bundle}$	2	3	15	15	59	91	156
	$Dive_{bundle}$	2	3	**14**	15	**52**	**75**	**114**
dh_6_15	E&C	10	35	209	338			
	Dive	5	14	95	147			
	$DE_{upperBound}$	5	69	495	1142	-	-	-
	$E\&C_{bundle}$	1	**3**	**16**	**21**	67	81	**138**
	$Dive_{bundle}$	1	4	19	25	**61**	**66**	183
dh_6_17_2	E&C	253	755	3345	-			
	Dive	58	153	725	1014			
	$DE_{upperBound}$	**0**	31	477	386	2159	-	-
	$E\&C_{bundle}$	7	8	33	49	89	182	303
	$Dive_{bundle}$	6	**7**	**24**	**24**	**48**	**121**	**178**
dh_6_16	E&C	21	69	141	214			
	Dive	13	36	71	97			
	$DE_{upperBound}$	12	111	777	1921	-	-	-
	$E\&C_{bundle}$	11	**24**	**29**	**42**	125	206	336
	$Dive_{bundle}$	11	25	37	54	**76**	**122**	**199**
dh_6_18	E&C	134	672	2140	3548			
	Dive	44	150	454	645			
	$DE_{upperBound}$	9	157	1790	3323	-	-	-
	$E\&C_{bundle}$	**7**	**11**	**19**	**29**	**120**	219	338
	$Dive_{bundle}$	9	20	24	31	131	**212**	**310**

Table 4.2: Time to prove optimality [sec]

Chapter 5

Recursive Evaluate and Cut

5.1 Introduction

The previous chapter presented improvements for the two-stage E&C algorithm. However, in reality, stochastic problems often have a structure where the uncertainty is revealed over time, and multiple decision stages are the result. Consider for example, the management of inventory, where new stock is ordered periodically. Another example where the uncertainty is observed over time and appropriate decisions are required periodically is the electric power generation and expansion planning; an optimal construction and generation plan of both new and existing power plants is required and regularly updated to meet future demand (Ahmed et al.; 2003). Previous chapters addressed two-stage stochastic programming, where problems are formalised under the assumption that the value of the random variables are observed at once. The scenario decomposition based E&C algorithm and extensions thereof were used to tackle two-stage problems. This chapter is looking at solving multi-stage combinatorial stochastic problems and will show how to extend two-stage E&C into a recursive multi-stage algorithm. At first, we illustrate how a naïve recursion of the algorithm looks like using a four-stage stochastic program. Thereafter, we present the new, improved version.

Contributions

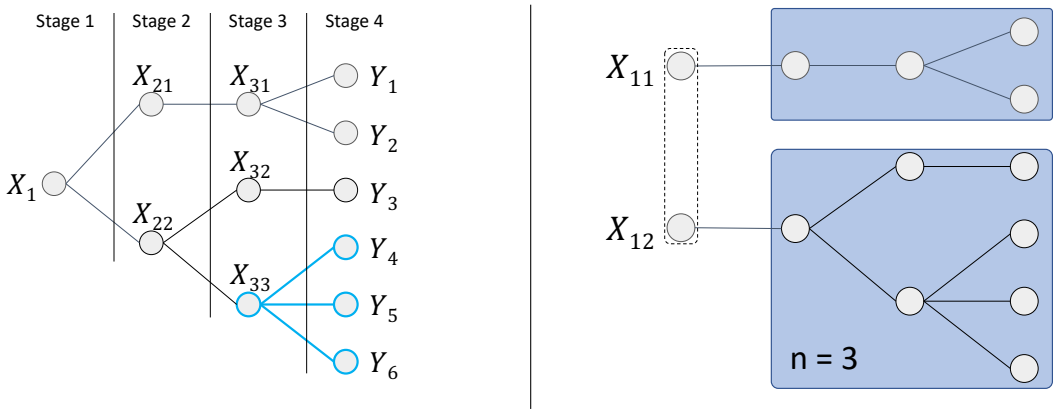
Previous chapters introduced E&C as an algorithm to solve two-stage stochastic problems with discrete first-stage variables. The main contribution of this chapter is the generalisation of E&C to multi-stage problems. Every algorithm for two-stage problems can be used to solve multi-stage problems if called recursively, however a naïve recursive approach may not exploit the problem structure effectively. We will present a new recursive algorithm that is different to the naïve approach as it makes use of shortcuts in the recursion to converge quickly to a feasible solution, while remaining complete. Computational results on a four-stage facility location problem with capacity, and other side constraints, demonstrates the performance of the algorithm. The contributions of this chapter have been published at the AAAI (Association for the Advancement of Artificial Intelligence) conference 2018 by Hemmi et al. (2018).

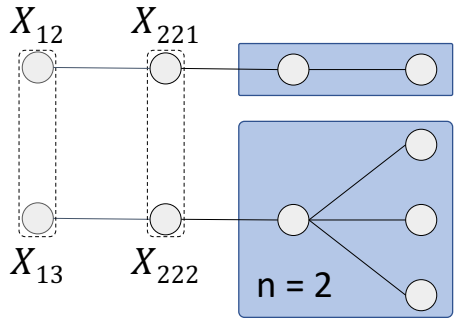
5.2 Recursive multistage E&C

5.2.1 Naïve E&C Recursion

This section describes the naïve recursion of E&C for multi-stage stochastic problems on a high level. Standard E&C requires two types of calls. Firstly, candidates are generated by solving the subproblems with enforced candidate nogoods, and secondly candidates are evaluated by projecting a variable assignment onto the shared variables before solving the subproblems. In standard E&C, the subproblems are deterministic optimisation problems (COP) and solved using a off-the-self MIP or CP solver. To tackle multi-stage problems with E&C, all that is required is a solver that is able to solve a $n-1$ subproblem, e.g. in a three-stage problem, a solver that is able to solve a two-stage problem.

To support the explanation or recursive E&C we use a four-stage minimisation problem as displayed in the picture below on the left. First, a copy of the shared variables is introduced for each of the subproblems, e.g. X_{11} and X_{12}. The four-stage SCOP is composed of two three-stage ($n=3$) subproblems. To generate candidates and to calculate a lower bound, the subproblems are solve to optimality. The resulting candidates are evaluated over the subproblems to obtain an upper bound, as in standard E&C. As long as the subproblems are also multi-stage SCOPs, the recursive algorithm calls itself to solve the $n-1$ subproblems.





In classical recursive fashion, the two $n-1$ stage subproblems are solved using the same methodology as just introduced. The three-stage suproblems are themselves composed of two subproblems with $n=2$. Finally, standard E&C is able to solve the resulting two-stage problems.

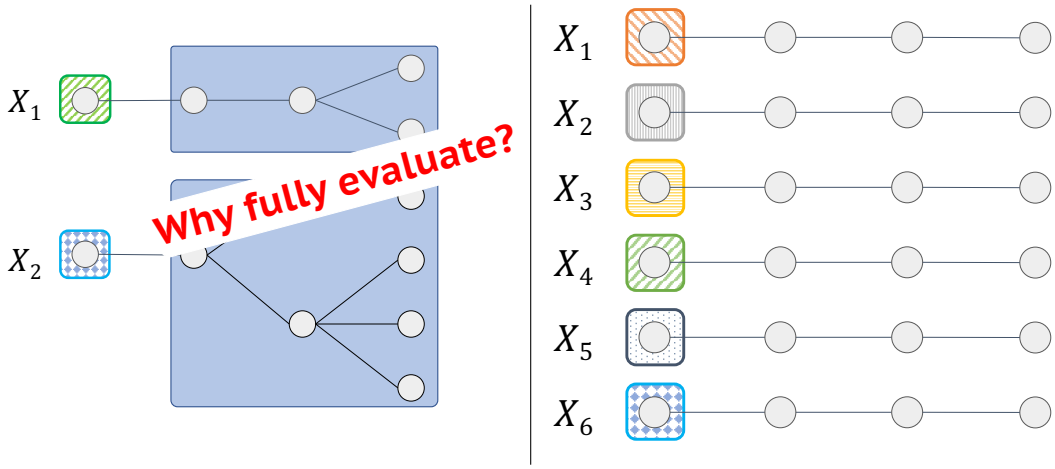
The naïve recursive E&C algorithm is guaranteed to find the optimal solution for combinatorial problems. However, it may take many iterations to even find the first set of $n=4$ candidates, as the subproblems have to be solved to optimality. In the following, we propose an improved recursive algorithm.

5.2.2 Improved E&C Recursion

In the previous section, we introduce the naïve E&C recursion. We now proceed with an improved version that requires fewer iterations, finds convergence quick and therefore solves the SCOP fast.

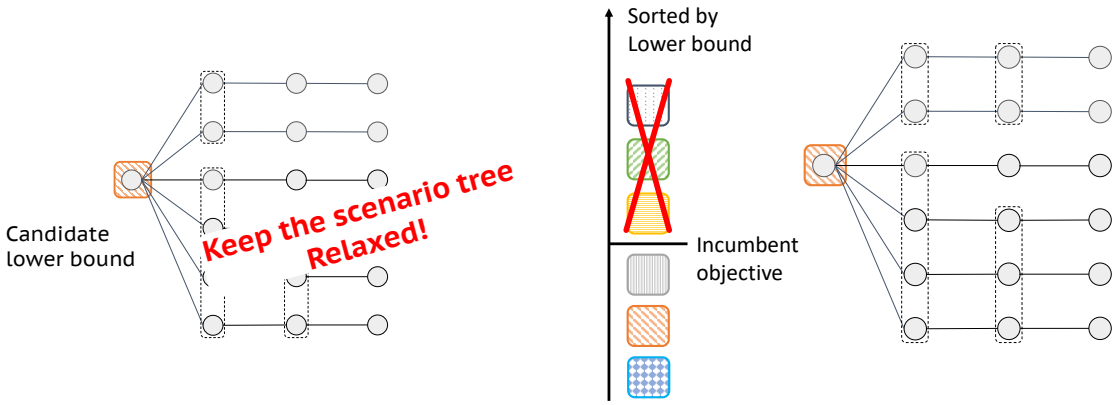
Standard E&C for two-stage problems requires that each scenario COP is solved to optimality and so does the naïve recursion; firstly, to obtain promising candidates, and secondly, to calculate the lower bound. In the two-stage case, the lower bound is only valid if the subproblems are solved to optimality. However, when solving multi-stage problems, the subproblems ($n > 1$) do not necessarily have to be solved to optimality; the difference will become apparent later.

The four-stage SCOP, as introduced before, is recursively decomposed until reaching the $n = 1$ level, see graph below on the right. In the naïve recursion, the next step would be to solve the two-stage ($n = 2$) problems to optimality, which provides in turn candidates for the $n = 3$ stage problem. This bottom-up approach is repeated until a candidate is found for the $n = 4$ stage problem. In contrast, we propose to not solve the subproblems bottom-up, but instead directly seek convergence in a top-down approach.



Solving each scenario COP immediately yields a valid lower bound on the four-stage SCOP, without having solved the subproblems to optimality. This lower bound is valid, in contrast to the two-stage case, as not enforcing the consistency constraints in the subproblems is a valid relaxation of the overall four-stage problem. *Note: The lower bound is monotonically increasing when enforcing more consistency constraints.*

Once a set of candidates for the $n = 4$ stage problem is obtained, we evaluate them over all scenarios of the current SCOP without enforcing the consistency constraints in the subproblems, see illustration below on the left. This yields a *candidate lower bound*, as the consistency constraints in the subproblems are not enforced. If a candidate lower bound exceeds the objective value of the incumbent solution, the respective candidate can be discarded immediately (picture bottom right). Otherwise, the procedure continues recursively with each $n - 1$ sub SCOP.



The advantage of our improved recursive E&C algorithm are the following:

- **Immediate lower bound**: After solving each scenario COP once, a lower bound for the original SCOP can be calculated immediately. In contrast, the naïve version requires to solve the subproblems to optimality before obtaining a lower bound.
- **Quick convergence:** A valid, converged solution is found after few steps, as we enforce consistency in a top-down approach.
- **Reject Candidates:** Candidates can be rejected very quickly, because of the candidate lower bound. There is no necessity to solve the subproblems to optimality before A) obtaining candidates and B) rejecting candidates.

We conceptually introduced our new and improved Recursive E&C algorithm. In the next section, we will introduce the algorithm formally.

Formal Algorithm

The implementation of the recursive E&C is written down in Algorithm 5. Firstly, the current SCOP $\mathcal{P}_{\mathcal{S}}$ is unpacked into all its scenarios COPs $(\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}k})$, in line 8. Solving the COPs (line 13) yields a lower bound on the current SCOP $\mathcal{P}_{\mathcal{S}}$ and a set of candidates S; σ_p is initially the empty assignment. Note, each candidate σ_v match the scope of the shared variables of the current SCOP $\mathcal{P}_{\mathcal{S}}$, denoted by $\sigma|_V$. In line 17, a method to obtain *candidate lower bounds* is called, which evaluates the candidates over the complete set of scenarios that belong to the current SCOP, e.g. $\mathcal{P}_{\mathcal{D}1} \ldots, \mathcal{P}_{\mathcal{D}k}$. This bound is a lower bound on the candidate, because the consistency constraints in the subproblems are not enforced. If the candidate lower bound (lb) is not strictly smaller than the incumbent, tested in line 20 , it is rejected immediately and a nogood is added to all subproblems $\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}k}$ in line 29. Otherwise, the algorithm proceeds in line 23 with solving the $n-1$ subproblems, with $\sigma_V \wedge \sigma_p$ projected onto the respective variables. The algorithm returns a policy tree as $\mathcal{T}$ as introduced in chapter 2.

Algorithm 5 Recursive Multistage Evaluate and Cut

```
1: procedure SOLVEMSCOP(P_S, σ_p)
2:     if P_S = P_D then
3:         < σ, obj > = SOLVE(P_D[C ∪= {σ_p}])
4:         T = < σ|_V, [] >
5:         return < T, obj >
6:     else P_S =< V, [P_S1, ..., P_Sn] >
7:         Initialize: UB = ∞
8:         [P_D1, ..., P_Dk] = GET_SCENARIOS(P_S)
9:         while LB < UB do
10:            % Obtain a lower bound and candidates
11:            LB = 0, S= ∅
12:            for i in 1..k do
13:                < σ,obj> = SOLVE(P_Di[C ∪= {σ_p}])
14:                LB += obj
15:                S ∪= {σ|_V}
16:            % Obtain candidate lower bound
17:            CLB = GETCANDIDATELB(S,[P_D1, ..., P_Dk],σ_p)
18:            % Evaluate candidates
19:            for (σ_V, lb) ∈ CLB ordered by lb do
20:                if lb < UB then
21:                    t_UB = 0
22:                    for i in 1..n do
23:                        < T_i, obj > = SOLVEMSCOP(P_Si, σ_V ∧ σ_p)
24:                        t_UB += obj
25:                    if t_ub < UB then
26:                        UB = t_UB
27:                        T =< σ_V, [T_1, .., T_n] >
28:                for i in 1..k do
29:                    ADDNOGOOD(P_Di, σ_V ∧ σ_p)
30:        return < T, UB >
31:
32: procedure GETCANDIDATELB(S,[P_S1, ..., P_Sk],σ_p)
33:    CLP = {}
34:    for σ_V ∈ S do
35:        lb = 0
36:        for i in 1..k do
37:            < _, obj > = SOLVE(P_Si[C ∪= {σ_V ∧ σ_p}])
38:            lb += obj
           CLB ∪= {(σ_V, lb)}
39:    return CLB
```

Memorisation:

At every recursion level, the candidates are evaluated against all the scenarios that match the scope of a given SCOP. At least one, and possibly many COPs already explored the variable assignment that matches that candidate, which results in redundant computations. This redundancy can be avoided by introducing a memory for each scenario $\mathcal{P}_{\mathcal{D}}$ that recalls whether a specific variable assignment has already been explored.

5.3 Experiments

To evaluate the performance of the proposed algorithm we are using two variations of the stochastic facility location problem with capacity constraints.

Multi-Stage Stochastic Facility Location Problem

The classical facility location problem has many applications, such as deciding where to open a warehouse in a supply chain, determining the best location of a database in a computer network, or selecting the most appropriate vendors when expanding into a new market (Arabani and Farahani; 2012). We motivate and evaluate our work using a multi-stage stochastic facility location problem with capacity constraints; Farahani et al. (2014) present a comprehensive review. The task is to open a number of facilities, assign each customer to a single facility, whilst minimising the combined cost of setting up facilities and distributing goods. The problem is composed of four stages and the number of customers (including locations) is uncertain and revealed over time. Each scenario describes a different set of customers.

Equation 5.1 contains the model for a single scenario and the complete model can be found in the appendix. The goal is to minimise the combined cost of setting up facilities and delivering goods to customers, where f_{ti} denotes the cost of setting up facility y_i in stage t and c_{tij} is the cost of delivering goods from facility i to customer j. Each customer is assigned to exactly one facility that might be different in each stage. C_i is the total capacity of facility i and d_j the demand of customer j.

The second variant of the problem contains an additional "balancing" constraint that enforces a similar customer count for each facility. The aim is to minimise the expected cost while satisfying the consistency constraints. For the experiments we generated 15 four-stage instances with 7 concrete realisations of the random variables in each stage, ending up with a total of 343 scenarios. For the 15 instances, we extracted further 9 problems, each contains a subset of the 343 scenarios, chosen such that a balanced tree results, e.g. 3 realisations in each stage yields 27 scenarios. We used a total count of 150 problems ranging from 27 to 343 scenarios for each of the problem classes. The instance parameters are: 6 facilities, 150 customers, the total warehouse capacity is 60% higher than the maximal customer demand, K_3 (the third-stage balancing constant) is set to 15 and the balancing constraints in stage one and two, K_1 and K_2, are set to 150. Essentially, the balancing constraint is only enforced in stage four.

We compare our algorithm with the performance of CPLEX on the DE. Both instances, the DE and the subproblems for Recursive E&C, are modelled in MiniZinc. The DE was solved using CPLEX (12.6.3) parallel optimiser and the Recursive E&C was implemented in Python 2.7 with CPLEX (12.6.3) as the subproblem solver. The experiments were carried out on a single computer that is part of the MonARCH HPC Cluster provided by Monash eResearch Centre with 16 physical cores (32 hyper threaded cores) at 3.20 GHz, with a time out of 1800 seconds per instance.

$$
\begin{aligned}
&\textbf{Basic model}\\
\min\Big\{ &\sum_{t\in T, i\in I} f_{ti} * y_{ti} + \sum_{t\in T}\sum_{i\in I, j\in J} c_{tij} * x_{tij} \Big\}\\
s.t. &\sum_{i\in I} x_{tij} = 1 && \forall j \in J, t \in T\\
&\sum_{j\in J} x_{tij} * d_j \le C_i && \forall i \in I, t \in T \qquad (5.1)\\
&x_{tij} \ge 0, y_{ti} = \{0,1\} && \forall t \in T, i \in I, j \in J\\
&\textbf{Balancing constraint}\\
&\Big|\sum_{j\in J} x_{ti_1 j} - \sum_{j\in J} x_{ti_2 j}\Big| < K_t && \forall i_1, i_2 \in I, t \in T
\end{aligned}
$$

5.4 Results

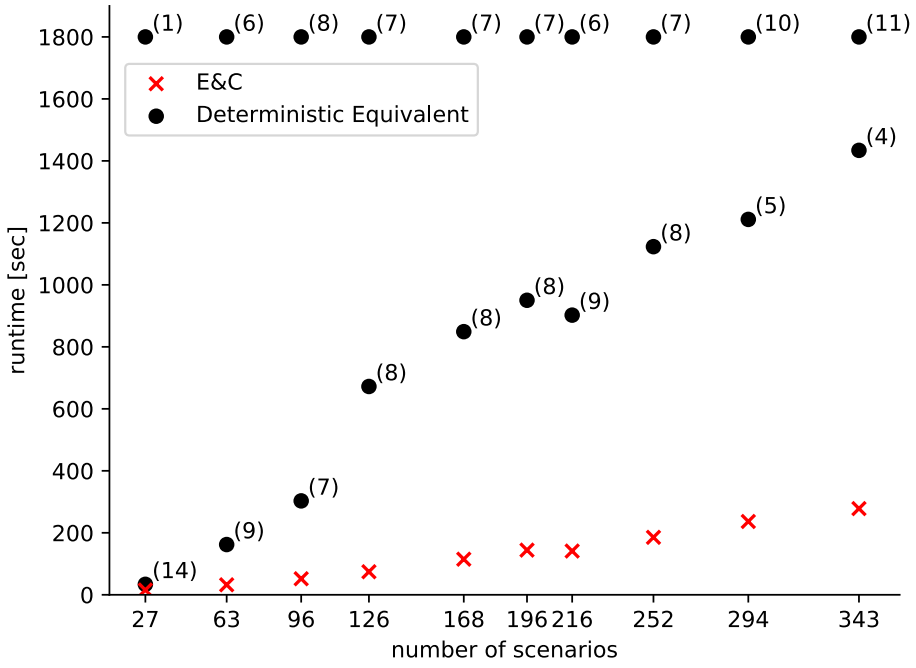
Basic Model:

Fig. 5.1 shows the time to solve the basic facility location problem using E&C and CPLEX on the DE. As expected, CPLEX is able to solve the DE for small instances in a reasonable amount of time but does not scale well when increasing the number of scenarios, as illustrated by the large number of instances that did not finish the search within the time out. For example, when looking at the instances with 343 scenarios, 11 out of 15 problems did not finish within the given 1800 seconds. In contrast, the E&C algorithm always finishes within the time out. The runtime of E&C increases approximately linearly with the number of scenarios per instance.

Model with extra constraint:

Fig. 5.2 shows the time to solve the facility location problem with balancing constraints using E&C and CPLEX on the DE. The additional constraints make it substantially harder for CPLEX to solve the DE, with only few instances solved to optimality within the time out. For most DE instances CPLEX cannot even determine an initial feasible solution (number in square brackets in Fig. 5.2). Proving optimality is a great challenge for CPLEX. This can even be observed when solving individual scenarios.

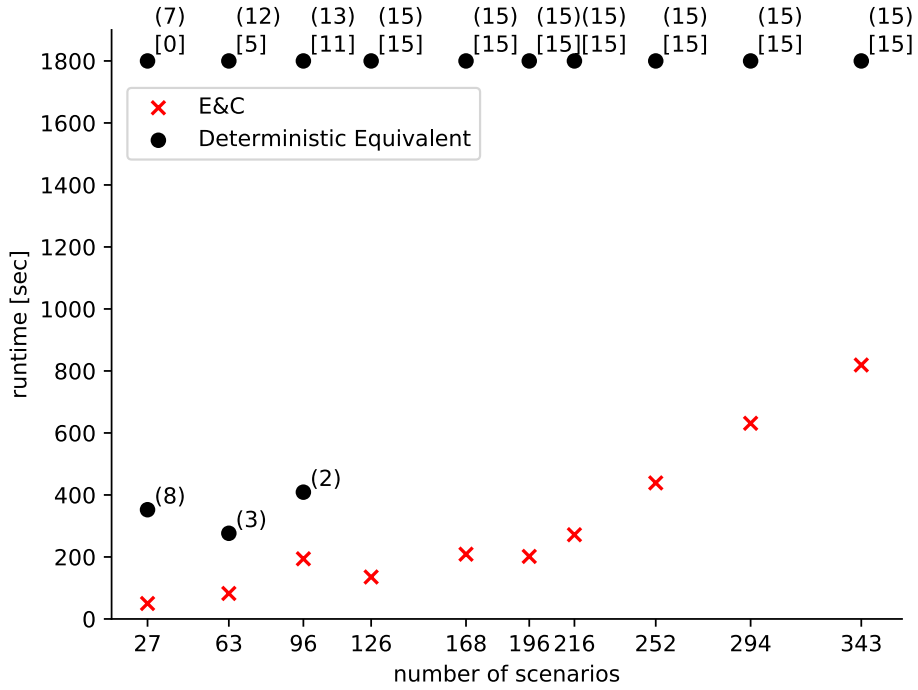
When solving subproblems in the E&C routine, we do not always require a full proof of optimality – all we need is a lower bound. Therefore, setting the relative optimality gap in CPLEX to 10^{-3} (instead of the default value of 10^{-4}) makes each scenario COP



- Average runtime of solving 15 instances grouped by number of scenarios.
- ([0-15]) top: number of instances that reach time-out
- ([0-15]) bottom: number of instances that are solved successfully
- Recursive E&C never reaches time-out.

Figure 5.1: Time vs. number of scenarios (basic model)

terminate much faster, without compromising on the completeness of the overall algorithm. The results in Fig. 5.2 clearly demonstrate the effectiveness of E&C. While CPLEX cannot cope with the DE formulation except in 13 out of 150 instances, recursive multistage E&C can find the optimal solution for all 150 instances within the time out.



- ([0-15]) top: number of instances that reach time-out
- ([0-15]) bottom: number of instances that are solved successfully
- [[0-15]] top: number of instances for which no integer solution was retrieved
- Recursive E&C never reaches time out.

Figure 5.2: Time vs. number of scenarios (extended model)

5.5 Related Work

Challenges in solving multi-stage stochastic optimisation problems with integer variables are well documented (Schultz; 2003). Various research directions have been explored, however a substantial amount of work has focused on decomposition algorithms. One line of work has developed algorithms that decompose stochastic problems by stages. An example of this decomposition is the L-shaped method (Van Slyke and Wets; 1969) inspired by Benders decomposition. Solver systems such as DECIS (Infanger; 1999) and FortSP (Ellison et al.; 2010) implement the nested L-Shaped method to solve multistage problems. However, those solvers are restricted to solve problems with linear constraints and continuous variables. In contrast, Recursive E&C can be used to solve problems with combinatorial structure.

Another line of work is based on the scenario decomposition. Watson et al. (2012) propose to use Progressive Hedging (PH) as a back-end solver for their modelling framework PySP. Rockafellar and Wets (1991) originally proposed the PH methodology to solve convex problems. The drawback of using PH is twofold; firstly, optimality is only guaranteed for continuous convex problems; and secondly, PH is not as flexible as Recursive E&C, as it requires parameter tuning that can be expensive in practice. Alonso-Ayuso et al. (2003) introduce an algorithm that relaxes the integrality requirements in addition to the consistency constraints, named Branch-and-Fix (BF). The scenarios are solved using a linear programming based branch-and-bound (b&b) procedure. To enforce consistency,

a common branching tree is used to fix nodes in the scenario b&b tree. Aldasoro et al. (2017) introduced a parallel BF coordination scheme that increases performance but does not remain complete. To the best of our knowledge, BF is not publicly available, it's implementation is not trivial and it is not clear how to implement it in a generic fashion to solve combinatorial problems.

In the context of multistage stochastic programming it is also worth mentioning dynamic programming, or stochastic dynamic programming (SDP) when applied to problems under uncertainty. When using SDP to model stochastic optimisation problems, the focus is typically on problems with a large number of decision stages or infinite horizons problems where the impact of future decisions is discounted (Birge and Louveaux; 2011). These problems are solved using either backward of forward recursion based algorithms or extensions thereof (Prestwich et al.; 2018). In the context of CP, dynamic programming has been addressed by Prestwich et al. (2018), who have shown that any discrete DP algorithm can be directly modelled and solved as a constraint satisfaction problem. Similar to the problems we are interested in, SDP models have discrete time steps and are modelled using discrete variables. In contrast, our work has focussed on exploring search algorithms that utilise standard MIP and CP solvers as back-ends to solve the subproblems. Furthermore, the problems studied in this chapter have a moderate number of decision stages.

Scenario clustering or bundling is a line of research orthogonal to the scenario decomposition that aims to tighten the bounds by bundling multiple scenarios into a DE. Clustering reduces the number of subproblems by strategically enforcing a subset of the consistency constraints throughout the entire search procedure. It capitalises on the fact that the DE can be solved efficiently for problems with a small number of scenarios. The effectiveness of scenario clustering has been demonstrated by Sandikci and Özaltın (2014), Aldasoro et al. (2017), and Escudero et al. (2016), amongst others. To incorporate scenario clustering into Recursive E&C is straightforward, however the focus of this chapter was on the introduction of the new algorithm.

5.6 Conclusion

The main contribution of this chapter is the generalisation of standard (two-stage) E&C to solve multi-stage stochastic problems. Unlike other algorithms, our method can be applied to solve problems with complex structure such as non-linear constraints and integer variables. No complicated problem reformulation is required to use Recursive E&C, making our method a prime candidate to be used as back-end solver for modelling frameworks such as GAMS, AMPL, AIMMS or Stochastic MiniZinc.

5.6.1 Limitations

Recursive E&C suffers from the same limitations as the standard two-stage algorithm, as it can only solve problems that produce candidates over discrete variables. Furthermore,

as shown in chapter 4, depending on the problem structure, many iterations are required to prove optimality when solving a two-stage stochastic problem with E&C.

5.6.2 Future Work

We demonstrated the effectiveness of Recursive E&C on two sets of benchmarks, by comparing E&C to solving DE formulation, the only available and applicable solver option. For future work, it will be interesting to investigate how information learned in one scenario, e.g. feasibility cuts, can be generalised and used in other scenarios, to improve the lower bound computations. Furthermore, ideas from logic-based benders decomposition (Hooker and Ottosson; 2003) and branch-and-check (Thorsteinsson; 2001) might lend themselves well to combine scenario decomposition with stage wise decomposition.

Chapter 6

Simulation Based Evaluate and Cut

6.1 Introduction

Until now, this thesis has been concerned with solving scenario based stochastic optimisation problems to optimality. This chapter will extend the context to problems that cannot easily be solved using a scenario based stochastic program as the number of scenarios required to represent the uncertainty appropriately is prohibitively large. First, we show that scenario-based SCOPs are not always the actual problem of interest, but rather an approximation thereof. Secondly, we present an algorithm that has been used widely with great success to solve stochastic problems. Thirdly, we propose a new algorithm and empirically demonstrate its effectiveness.

We will now use an example to show that the uncertainty model used for scenario based stochastic programs is often an approximation of the random variables. Consider a two-stage facility location problem as displayed in Fig. 6.1, with uncertainty around the available set of customers. In the first stage, we decide where to open distribution centres, also called facilities, and in the second stage, once the set of available customers is known, we assign them to the open facilities. After reaching out to all potential customers we have a list of locations and commitment statements. Each prospective customer indicates that the likelihood of opening a shop is 50% irrespective of what all others decide (a set of independent probabilities). There are 20 possible shop locations and therefore, to completely capture the uncertainty, about one million scenarios (2^{20}) are required. Clearly, as the number of customers grows it becomes impossible to consider the complete set of scenarios and a subset must be chosen to approximate the random variables.

In reality, the number of scenarios required to sufficiently approximate the problem is often much lower than a million. Wu et al. (2017) demonstrate, using a simulation approach, that a few thousand scenarios ($\sim$3000) sufficiently approximate the uncertainty in the road network investment problem they study. However, Wu et al. (2017) are not able to solve problems with more than 30 to 200 scenarios to optimality.

Generating scenarios from random distributions is an important topic in stochastic programming. However, we will not address how to sample scenarios as this is a field of

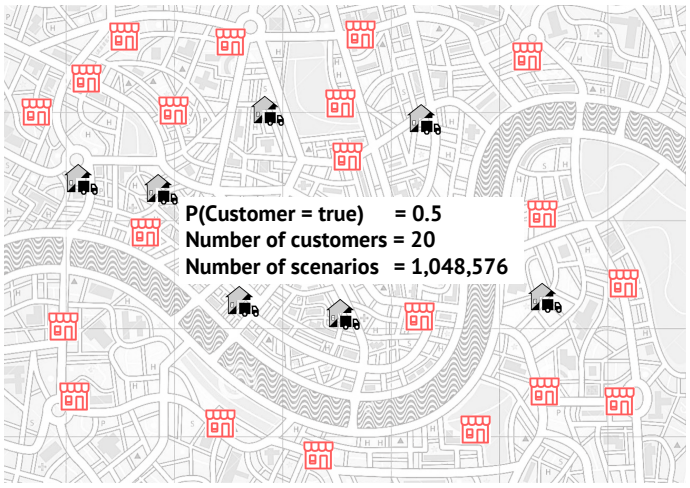


Figure 6.1: A two-stage facility location problem

research by itself. There is a vast body of research regarding scenario generation (Vázsonyi; 2006; Høyland and Wallace; 2001) and reduction techniques to select a subset of all scenarios (Dupačová et al.; 2003; Heitsch and Römisch; 2003). For the rest of this chapter, we assume that the set of scenarios is given and focus on how to solve the stochastic problem.

The time to solve a SCOP generally depends on the number of scenarios considered, as seen in previous chapters. The graph in Fig. 6.2 conceptually describes the relationship between the number of scenarios and the time to solve the resulting SCOP. The super linear function (purple) indicates that solving SCOPs with a relatively low number of scenarios can be done effectively, however as the scenario count increases it becomes increasingly challenging to solve the SCOP. In contrast, the linearly increasing line (green) describes the time to evaluate or simulate the quality of a given candidate over a set of scenarios. The characteristics of the two graphs suggests that a small number of scenarios can be considered to solve an optimisation problem and a large number of scenarios can be used to simulate the objective value of a given policy.

Kleywegt et al. (2002) proposed an approach, that combines optimisation with simulation techniques, to solve SCOPs with a large number of scenarios. The idea is straightforward and displayed in Fig. 6.2; firstly, a small number of scenarios is sampled, and the resulting SCOP is solved. Secondly, the obtained candidate, the optimal solution to the SCOP, is evaluated over a much larger set of scenarios to approximate the objective function more accurately. The two steps are repeated until a given stopping criteria is satisfied. In this chapter we propose an algorithm that is inspired by the work of Kleywegt et al. (2002) but intended to solve problems that are computationally more challenging than what the algorithm by Kleywegt et al. (2002) can solve.

Contributions

The main contribution of this chapter is called Simulation Based Evaluate and Cut, a method that augments standard E&C with simulation, to solve stochastic problems that

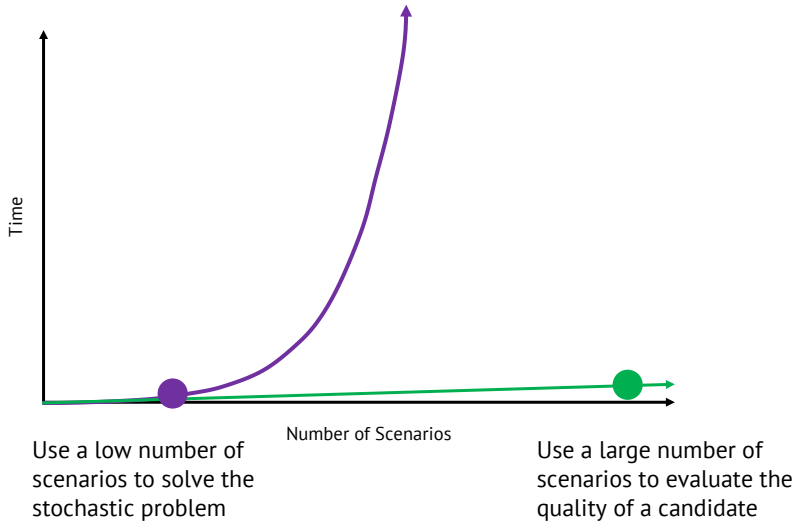


Figure 6.2: Time to solve a stochastic problem

require a large number of scenarios for describing the random variables. The chapter starts with a background section that introduces the Monte Carlo sampling based algorithm by (Kleywegt et al.; 2002). Thereafter, we show how to combine standard E&C with simulation to end up with an algorithm that performs strongly at solving stochastic problems with a large number of scenarios. Simulation based E&C is developed with very challenging decision problems in mind, where even a SCOP with fewer than ten scenarios cannot be solved to optimality. To evaluate the performance of our new algorithm we use the stochastic template design problem introduced in chapter 3 and compare it to the method proposed by (Kleywegt et al.; 2002).

6.2 Background - Monte Carlo Simulation

Kleywegt et al. (2002) proposed a Monte Carlo sampling based approach, called Sample Average Approximation method, to solve stochastic discrete optimisation problems of the following form.

Definition 6 *The **true** or real stochastic problem is defined as:*

$$\begin{aligned} g(x) &= \mathbb{E}\, G(x,S) \\ &\underset{x\in D}{\arg\min}\ g(x) \end{aligned} \tag{6.1}$$

Where S are the stochastic variables drawn from a probability distribution P; D is the finite domain of x; $G(x,S)$ is a real valued function of the two (vector) variables x and S; and $\mathbb{E}\, G(x,S) = \int G(x,s)P(ds)$ is the corresponding expected value. The random variables S have either continuous or infinite discrete support. We assume that the expected value function $g(x)$ is well defined, i.e., for every $x \in D$ the function $G(x,\cdot)$ is measurable and $\mathbb{E}\{G(x,S)\} < \infty$. Furthermore, we define $v* = min_{x\in D} g(x)$ to be the optimal solution.

The main features of the problem we study are:

1. We only study two-stage problems.
2. The expected value problem $g(x) = \min_{x \in D} \mathbb{E}\, G(x, S)$ cannot easily be solved, e.g. no closed-form is available, and the number of scenarios used to approximate the random variables is prohibitively large.
3. Given x and s the function $G(x, s)$ reduces to a COP $\mathcal{P}_{\mathcal{D}}$ and the objective can easily be computed. It is therefore possible to evaluate a candidate over a large number of scenarios.
4. It is possible to generate an iid (independent identically distributed) set of samples from the random variables S.
5. The problem has relatively complete recourse. In other words, for every feasible $x \in D$ the second-stage problem is also feasible; $\mathbb{E}\{G(x, S)\} < \infty$. This is important as we approximate the quality of a candidate over a non-complete set of scenarios. An infeasible second stage would yield an infinite objective value and therefore an infinite variance.

6.2.1 Sample Average Approximation Method

The basic idea behind the sample average approximation method as introduced by Kleywegt et al. (2002), is straightforward. First, a set of samples of S is generated to approximate the original problem. The sampled SCOP is then solved to obtain a candidate, which is evaluated over a large number of scenarios to approximate the objective more accurately. The two steps are repeated until a stopping criterion is satisfied.

A sampled SCOP is defined as:

Definition 7 *A two-stage **sample average approximation** (SAA) problem is a pair:*

$$\hat{\mathcal{P}}_{\mathcal{S}} \quad =< V, [\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}N}] >$$

with one set of shared variables V and N iid sampled scenarios, $\mathcal{P}_{\mathcal{D}1}, \ldots, \mathcal{P}_{\mathcal{D}N}$.

Solving $\hat{\mathcal{P}}_{\mathcal{S}}$ yields a candidate that can be evaluated over a larger number of scenarios to get a better approximation of the objective value.

6.2.2 Evaluating Candidate Solutions

To evaluate the quality of a candidate, Kleywegt et al. (2002) proposes to calculate the optimality gap between the objective value of the candidate and the "true" optimal objective value with statistical confidence. To calculate the optimality gap, an estimate of the "true" optimal objective value is required and an estimate of the candidate objective value. Given feasible solution $\hat{x} \in D$, a candidate, we now show how to calculate the optimality gap.

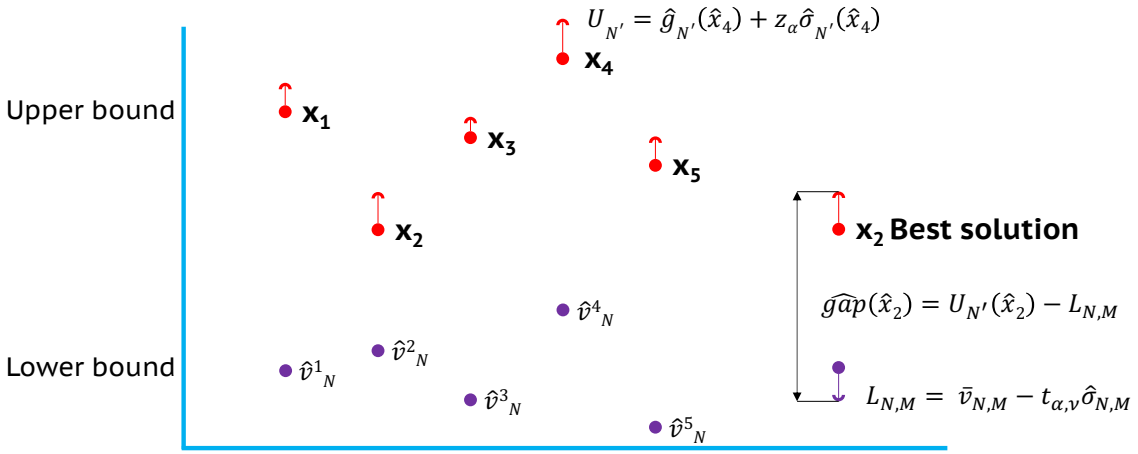


Figure 6.3: SAA illustration

For every $\hat{x}$, it is known that $g(\hat{x}) \geq v^*$ given that v^* is the "true" optimal objective of 6.1. The quality of any given candidate $\hat{x}$ can be measured by the optimality gap

$$gap(\hat{x}) = g(\hat{x}) - v^*.$$

However, since v^* cannot be calculated, a statistical procedure to estimate its quality is required.

Optimality Gap - Upper Bound

First, we show how to estimate the value of $g(\hat{x})$ using Monte Carlo sampling, with a random set of iid samples $\mathcal{P}_{\mathcal{D}_j}$ with $j = 1, \ldots, N'$ of S, which results in a SCOP $\hat{\mathcal{P}}_{\mathcal{S}N'}$. The value of the candidate can be estimated by

$$\hat{g}_{N'}(\hat{x}) \quad = N'^{-1} \sum_{j=1}^{N'} f_j(\hat{x})$$

where $f_j(\hat{x})$ is the resulting objective value of evaluating the candidate over scenario j and the variance is,

$$\hat{\sigma}^2_{N'}(\hat{x}) \quad = \frac{1}{N'(N'-1)} \sum_{j=1}^{N'} [f_j(\hat{x}) - \hat{g}_{N'}(\hat{x})]^2.$$

The sample size N' can be large as evaluating a candidate $\hat{x}$ over a set of scenarios requires only solving the second-stage problem. The value of a candidate is estimated by calculating the one-sided confidence interval (Colquhoun; 1971),

$$U_{N'}(\hat{x}) = \hat{g}_{N'}(\hat{x}) + z_\alpha \hat{\sigma}_{N'}(\hat{x}),$$

which provides an approximate $100(1-\alpha)$ confidence upper bound for $g(\hat{x})$. The one-sided confidence interval, as opposed to the two-sided, is chosen because we are interested in stating the confidence of a value being below a certain value, the upper bound. Kleywegt

et al. (2002) argues that this bound is justified by the Central Limit Theorem, which states that given a sufficiently large number of iid samples, the resulting distribution will approximately follow a normal distribution. To calculate the upper-tailed test, we require the critical value z_α, e.g. for $\alpha = 5\%$ we get $z_\alpha \approx 1.64$ and for $\alpha = 1\%$ we get $z_\alpha \approx 2.33$ (Maddala and Lahiri (1992) provides tables with corresponding values for z_α). *A critical value is the point on the scale of the test statistic beyond which the null hypothesis is rejected.* A concrete example of calculating the upper bound for a set of candidates is illustrated in Fig. 6.3. The upper bound $U_{N'}$ is calculated for each candidate, $x_1, \ldots, x_5$.

Optimality Gap - Lower Bound

The result of evaluating the set of candidates obtained by solving the SAA problems yields an upper bound. We now proceed with an estimator for v^*. Let us define $\hat{v}_N$ as the optimal objective of the sampled SCOP $\hat{\mathcal{P}}_{\mathcal{S}N}$, as illustrated in Fig. 6.3 on the bottom with $\hat{v}_N^1, \ldots, \hat{v}_N^5$. Note, $\hat{v}_N$ is a function based on a set of random samples and is therefore random itself.

The sample average $\hat{g}_N(x)$ is an unbiased estimator of the expectation of $g(x)$ and therefore $\mathbb{E}[\hat{g}_N(x)] = g(x)$ holds.

$$g(x) = \mathbb{E}[\hat{g}_N(x)] \geq \mathbb{E}[\hat{v}_N]$$

When minimising the left-hand side of the above inequality we obtain that $v^* \geq \mathbb{E}[\hat{v}_N]$.

To estimate $\mathbb{E}[\hat{v}]$ we solve multiple SAA problems and average the resulting optimal values, as in Fig. 6.3 on the bottom right. We generate M SAA problems based on iid samples, each of size N and solve them to optimality. Let $\hat{v}_N^1, \ldots, \hat{v}_N^M$ be the matching optimal values. Then

$$\bar{v}_{N,M} = \frac{1}{M} \sum_{j=1}^{M} \hat{v}_N^j$$

is an unbiased estimator of $\mathbb{E}[\hat{v}_N]$ with the variance

$$\hat{\sigma}_{N,M}^2 = \frac{1}{M(M-1)} \sum_{j=1}^{M} (\hat{v}_N^j - \bar{v}_{N,M})^2$$

As previously to estimate the upper bound, we calculate the lower bound with a confidence interval using a one-tailed test to determine $\mathbb{E}[\hat{v}_N]$ with $100(1-\alpha)\%$ confidence. When calculating the confidence interval for the upper bound, we used the z-score, which was justified by a large sample, e.g. N'. However, since the number of samples to calculate the confidence interval for the lower bound is in practice often small, e.g. $M = 10$, using the t-test is correct (Colquhoun; 1971).

$$L_{N,M} = \bar{v}_{N,M} - t_{\alpha,\nu} \hat{\sigma}_{N,M}$$

where $\nu = M - 1$ and $t_{\alpha,\nu}$ is the α-critical value of the t-distribution with ν degrees of freedom. *The degrees of freedom ν is equal to the number of samples, e.g. M, minus one.*

Since $v^* \geq E[\hat{v}_N]$, we have that $L_{N,M}$ provides a valid statistical lower bound on v^*, consequently

$$\widehat{gap}(\hat{x}) = U_{N'}(\hat{x}) - L_{N,M} \tag{6.2}$$

gives a statistically valid bound on the true gap, with confidence $1 - 2\alpha$.

SAA Algorithm

The previous section introduced the concept of a sampled SCOP and how to estimate an optimality gap. The resulting algorithm as proposed by Kleywegt et al. (2002) is displayed in Alg. 6. First, the size of N, N', M and a stopping criteria are defined. Kleywegt et al. (2002) propose a formula to determine N that is based on problem specific parameters such as the dimension of the solution space and the variance of the approximated objective function. They admit that for many problems it is not easy to compute the size of N and even if possible Wu et al. (2017) mention that in practice the resulting size of N can easily grow to become prohibitively large. However, a small, well chosen set of samples is often sufficient. The size of N' can exceed N substantially as evaluating a candidate over a set of scenarios is cheap compared to finding the optimal solution for the same scenarios. No strict rule to determine M is given, yet Shapiro and Philpott (2007) argue that in practice 5 to 10 replications are sufficient. The stopping criteria can be defined using the estimator gap (6.2) or its variance. If the stopping criteria is not satisfied, then N or M is increased.

After all parameters are defined a sampled SCOP is generated in line 4 and solved in line 5. The result is a first estimator $\hat{v}_N^1$ and candidate x_1, as in Fig. 6.3 on the left side. The upper bound $U_{N'}$ for x_1 is then calculated for estimating the optimality gap $\widehat{gap}$ in line 6. If the stopping criterion is not satisfied after repeating these steps M times, either N or N' are increased and the algorithm starts over again.

Algorithm 6 Monte Carlo based SAA

1: **procedure** SOLVESAA
2: Choose: N, N', M and a stopping criteria
3: **for** m in 1,...,M **do**
4: sample N scenarios and generate $\hat{\mathcal{P}}_{\mathcal{S}_N^m}$
5: $< \hat{x}_N^m, \hat{v}_N^m >= solve(\hat{\mathcal{P}}_{\mathcal{S}_N^m})$
6: estimate gap $\widehat{gap}(\hat{x}) = U_{N'}(\hat{x}_N) - L_{N,1..m}$
7: if stopping criteria is satisfied, go to line 9
8: If stopping criteria is not satisfied, increase N and/or N' and go back to line 4
9: Choose the best solution $\hat{x}$ among all candidate solutions $\hat{x}_N^m$.

6.2.3 Discussion

Monte Carlo simulation-based methods have been applied successfully to solve stochastic problems, see Shapiro et al. (2009); Kim et al. (2015). However, in practice the sample size N is often smaller than what the calculations proposed by Kleywegt et al. (2002) would suggest. On the one hand, Sheldon et al. (2012) and Wu et al. (2017) argue and experimentally demonstrate that a small (e.g. < 30) yet well representative set of samples

is sufficient to find good policies. On the other hand, solving SCOPs even with a small number of scenarios is often expensive or not feasible; Wu et al. (2017) use the SAA scheme to solve a stochastic network design problem and report a runtime of 2.5 hours to solve a SCOP with 20 scenarios. In addition, finding representative scenarios is a challenge by itself. In the next section we propose an algorithm that does not require solving a SCOP and therefore scales to problems that cannot be solved by the method introduced by Kleywegt et al. (2002).

6.3 Simulation-based Evaluate and Cut

We now proceed with the main contribution of this chapter, which is a method that combines E&C with the simulation based Monte Carlo method. As in standard E&C, candidates are generated by solving scenarios. In contrast to the SAA method as introduced above, we generate a large number of candidates. To evaluate the quality of the candidates, we propose a three step procedure that is able to eliminate weak candidates quickly and evaluate promising candidates over a large number of scenarios.

Our new algorithm is based on the following observations. Firstly, Kleywegt et al. (2002) have shown that solving an SAA with a low number of scenarios yields promising candidates for the true stochastic problem. Secondly, when solving a SCOP with E&C, the optimal or a near optimal solution is often retrieved in the first iteration. If this assumption holds true, all remaining iterations of the E&C scheme are only required to prove that the incumbent candidate is optimal. Now, instead of solving the SAA problem ($\hat{\mathcal{P}}_{\mathcal{S}_N^m}$) to optimality, we directly use the set of candidates obtained in the first iteration of E&C as candidates for the true stochastic problem and obtain candidates of equal quality compared to the method by Kleywegt et al. (2002). The subsequent sections formalise our intuition and provide empirical evidence.

6.3.1 Algorithm

The algorithm we propose is composed of two parts. First, we generate candidates and secondly the quality of the candidates is approximated using an iterative simulation approach.

Candidate generation

As in the E&C algorithm, individual scenarios are used for obtaining candidates. However, in contrast to E&C, where nogoods are added only after the first iteration, which includes the evaluation phase, we add candidate nogoods directly to a pool after generating them. This is done, because we are interested in obtaining a large set of candidates without any duplicates.

Algorithm 7 illustrates how to generate candidates. A scenario based stochastic problem $\mathcal{P}_{\mathcal{S}_{generate}}$, or SAA, is given as input. In line 2 we define an empty set of candidates and specify the number of candidates to be generated. Thereafter, in line 4 to 9 the requested number of candidates is generated. First, in line 6, we add the already generated

candidates as nogoods to the next scenario $\mathcal{P}_{\mathcal{D}}$, before solving it. The first-stage variable assignment is then added to the set of candidates S and the objectives are summed up to compute a lower bound. The stopping criteria can also be replaced by other measures, for example total time passed. *Note: If M_1 is greater than the number of scenarios in $\mathcal{P}_{\mathcal{S}_{generate}}$, multiple candidates can be generated from the same scenario, until the size of S is equal to M_1.*

The next section will illustrate how to evaluate the quality of the candidates.

Algorithm 7 Generate Candidates

```
1:  procedure GENERATECANDIDATES(P_{S_generate})
2:      Initialize: S = ∅, LB = 0, M_1
3:      [P_{D_1},...,P_{D_k}] = GET_SCENARIOS(P_S)
4:      for i in 1..M_1 do
5:          for σ_V ∈ S do
6:              ADDNOGOOD(P_{D_i},σ_V)
7:          < σ,obj> = SOLVE(P_{D_i})
8:          S ∪= {σ|_V}
9:          LB += obj
10:     return <S,LB>
```

Candidate evaluation

To evaluate the quality of the candidates effectively, we propose an iterative ranking method. Kleywegt et al. (2002) proposed to evaluate a small number of candidates over a large number of scenarios. This approach does not work well for Simulation Based E&C as the number of candidates M_1 can be very large. Therefore, we propose to simulate all candidates in S over a small number of scenarios, rank them by objective and reject all candidates that are not promising.

Algorithm 8 describes the simulation procedure. A sampled SCOP $\mathcal{P}_{\mathcal{S}_{evaluate}}$ is used to evaluate the candidates. First, in lines 5 to 8, a $\mathcal{P}_{\mathcal{S}_{evaluate}}$ with a small number N_1 of scenarios is used for the evaluation. The candidates and resulting objectives are added to a map, which is ordered by objective value. As the scenarios are chosen randomly, the resulting objective is an unbiased estimator of the true objective value. Thereafter, in lines 11 to 16, the highest ranked subset (size M_2) is chosen for further evaluation and the remaining candidates are discarded. The objective value obtained in the second evaluation phase is combined with the previously calculated objective and added to the candidate map, line 16. Lastly, in line 19 to 24, a small subset (M_3) of candidates is evaluated over a much larger number of scenarios to reduce the variance of the objective value.

Algorithm 8 Evaluate Candidates

1: **procedure** EVALUATECANDIDATES($\mathcal{P}_{\mathcal{S}_{evaluate}}$, S)
2: Initialize: cnd_ranking = $\varnothing$
3: **% extract a SCOP with a low number of scenarios to**
4: **% evaluate a large number of candidates**
5: $\mathcal{P}_{\mathcal{S}_{small}}$ = GET_SUBPROBLEM($\mathcal{P}_{\mathcal{S}}$, N_1)
6: **for** $\sigma_V \in$ S **do**
7: < _, obj> = SOLVE($\mathcal{P}_{\mathcal{S}_{small}}$[C $\cup =$ $\{\sigma_V\}$])
8: cnd_ranking.add(<obj, σ_V >)
9: **% extract a SCOP with a medium number of scenarios to**
10: **% evaluate a small number of selected candidates**
11: S = $\varnothing$
12: S = get_m_best_candidates(cnd_ranking, M_2)
13: $\mathcal{P}_{\mathcal{S}_{medium}}$ = GET_SUBPROBLEM($\mathcal{P}_{\mathcal{S}}$, N_2)
14: **for** $\sigma_V \in$ S **do**
15: < _, obj> = SOLVE($\mathcal{P}_{\mathcal{S}_{medium}}$[C $\cup =$ $\{\sigma_V\}$])
16: cnd_ranking[σ_V].add(obj)
17: **% extract a SCOP with a large number of scenarios to**
18: **% reduce the objective variance on the few best candidates**
19: S = $\varnothing$
20: S = get_m_best_candidates(cnd_ranking, M_3)
21: $\mathcal{P}_{\mathcal{S}_{large}}$ = GET_SUBPROBLEM($\mathcal{P}_{\mathcal{S}}$, N_3)
22: **for** $\sigma_V \in$ S **do**
23: < _, obj> = SOLVE($\mathcal{P}_{\mathcal{S}_{large}}$[C $\cup =$ $\{\sigma_V\}$])
24: cnd_ranking[σ_V].add(obj)
25: **return** cnd_ranking.get_best()

6.3.2 Experiments

For the empirical evaluation of Simulation Based E&C, we use the stochastic template design problem as introduced in section 3 in Fig. 3.5. The complete model can be found in the appendix, Fig. A.1.2. Hereafter, the lines refer to Fig. 3.5 or Fig. A.1.2 as the lines are consistent across both models. To recap; in the first stage (line 22), we design a number of templates that will later be used to print animal food packages; in the second stage (line 25), after observing demand, we decide how many times to print each template. The goal is to minimise the expected number of prints (line 74) while satisfying the demand (line 40). This problem complies with the five features defined earlier:

1. It is a two-stage problem.
2. The expected value problem $g(x) = \min_{x \in D} \mathbb{E}\, G(x, S)$ is difficult to solve.
3. Evaluating a candidate is cheap. Given a first-stage assignment, the problem becomes linear and the LP relaxation is tight.

4. It is possible to generate an iid set of samples.

5. The problem has complete recourse as the parameters are chosen accordingly. This means in practice that the demand of each package type is always greater than zero.

We designed two types of benchmarks. Both the model and the instances can be found in the appendix. The parameters to generate the benchmarks are chosen such that the resulting instances have a similar scenario count to problems solved by others including Wu et al. (2017), who have solved a real-world road network investment problem.

Two-template (easy) benchmark

This is an easy benchmark where a $\mathcal{P}_{\mathcal{S}}$ with a moderate number of scenarios, e.g. 30, can be solved within an hour. Two templates, each with nine slots, must be designed to print six different food packages. For each food type we drew four random numbers. The number of scenarios required to represent all possible combinations is 4096, too many to be solved optimally.

Three-template (difficult) benchmark

This is a difficult benchmark where a $\mathcal{P}_{\mathcal{S}}$ even with a small number of scenarios, e.g. 2, cannot be solved to optimality within an hour. Three templates, each with nine slots, must be designed to print five different food packages. For each food type we drew five random numbers. The number of scenarios required to represent all possible combinations is 3125.

6.3.3 Results

We compare our new algorithm, Simulation Based E&C, with the established Monte Carlo based SAA algorithm as introduced earlier. In Table 6.1 we display the relevant results.

Monte Carlo based SAA

We used the easy benchmark (with two templates) to assess the impact of N, the number of scenarios used to construct the SAA $\hat{\mathcal{P}}_{\mathcal{S}N}$. The parameter N impacts the quality of the retrieved candidates and the optimality gap.

To obtain the SAA's, we sampled four times ten (M=10) $\hat{\mathcal{P}}_{\mathcal{S}N}$ replications with (N) 10, 20, 30 and 50 scenarios. The $\hat{\mathcal{P}}_{\mathcal{S}N}$ are then solved using the deterministic equivalent and CPLEX 12.62 and the resulting candidates (≤ 10) are evaluated over the complete set of 4096 scenarios.

Preliminary results have shown that E&C does not perform well on the template design instances, while E&C finds an optimal or near optimal candidate quickly, proving optimality requires many iterations. The ten $\hat{\mathcal{P}}_{\mathcal{S}N}$ with 50 scenarios could not be solved within the time out of 1 hour and we therefore did not include them in the results.

The difficult benchmark with three templates turned out to be too hard to be solved using the SAA method. Even a $\hat{\mathcal{P}}_{\mathcal{S}N}$ with two scenarios could not be solved within 1 hour.

On average an optimality gap of about 30% remained. Therefore, we cannot report any results for the difficult benchmark in Table 6.1.

The grey columns in Table 6.1 clearly confirm that SAA, when applicable, works extremely well even with a low N. The optimality gap, especially with $N = 30$ is tight.

Simulation Based E&C

To analyse the performance of Simulation Based E&C we conducted multiple experiments. Firstly, we retrieved 1000 candidates using the easy benchmark and evaluated all of them over the complete set of 4096 scenarios. As seen in the right column in Table 6.1, the quality of the candidates matches SAA. However, as expected, the optimality gap is not as tight compared to the Monte Carlo based SAA method.

Secondly, to assess how well the order of candidates (ordered by objective value) can be approximated using simulation, we evaluated each candidate over a small subset of samples, ranked them and compared the ranking with the "real" ranking, the result of evaluating each candidate over the complete set of samples. The result of comparing the simulated rankings is displayed in Fig. 6.4. First, each candidate is evaluated over the complete scenario set (4096) and sorted by objective value. Secondly, the candidates are evaluated over a randomly chosen subset of 10, 20, 30, 50 and 100 scenarios and sorted by objective value. The indices of each ranking are then displayed using a scatter plot. Plotting a perfectly approximated ranking would produce a diagonal with an increasing slope of 45 degrees as displayed in yellow; the simulated and true ranking would match. Clearly, the accuracy of simulating the ranking increases when more scenarios are used, however already a relatively small number (<100) of scenarios is sufficient to approximate the ranking of candidates relatively well.

Choosing the parameters N_1 and M_2 in Algorithm 8 appropriately is important as otherwise the most promising candidates are discarded due to a poor approximation of the ranking or a too aggressive candidate rejection. It is essential to determine the number of scenarios that are required for the initial simulation and the number of candidates to keep afterwards. In Fig. 6.5 we print the simulated index of the best candidate. For example, a data point with x-value 10 and y-value 100 shows that the best candidate of an instance when simulated using 10 scenarios was ranked 100 in simulation. Clearly, when choosing $N_1 > 30$ it is sufficient to keep a few hundred candidates for the second evaluation iteration.

Runtime

We have shown that Simulation Based E&C produces candidates of similar quality compared to the SAA method for the easy benchmarks and that SAA fails on the difficult problems, as it is impossible to solve the stochastic problem. In this paragraph we will relate the above results to execution times. The relevant execution times are displayed in Table 6.2. As mentioned before, preliminary results indicated that E&C is not competitive in solving the template design problem, as the number of iterations required to prove optimality is large. As expected, the time to solve a deterministic equivalent model of

		SAA			E&C$_{statistical}$
	# scenarios	10	20	30	
template_design_easy_a	upper bound	574	566	566	562
	lower bound	523	549	554	440
	gap [%]	8.8	3.0	0.35	21.7
template_design_easy_b	upper bound	557	557	557	563
	lower bound	534	550	551	440
	gap [%]	4.1	1.2	1.1	21.8
template_design_easy_c	upper bound	560	553	553	553
	lower bound	535	542	547	462
	gap [%]	4.4	1.9	1.1	16.4
template_design_easy_d	upper bound	594	591	591	598
	lower bound	564	580	587	488
	gap [%]	5.1	1.9	0.7	18.3
template_design_difficult_a	upper bound	-	-	-	455
	lower bound	-	-	-	413
	gap [%]	-	-	-	9.2
template_design_difficult_b	upper bound	-	-	-	433
	lower bound	-	-	-	375
	gap [%]	-	-	-	13.3

Table 6.1: Comparison of SAA method and Statistical E&C

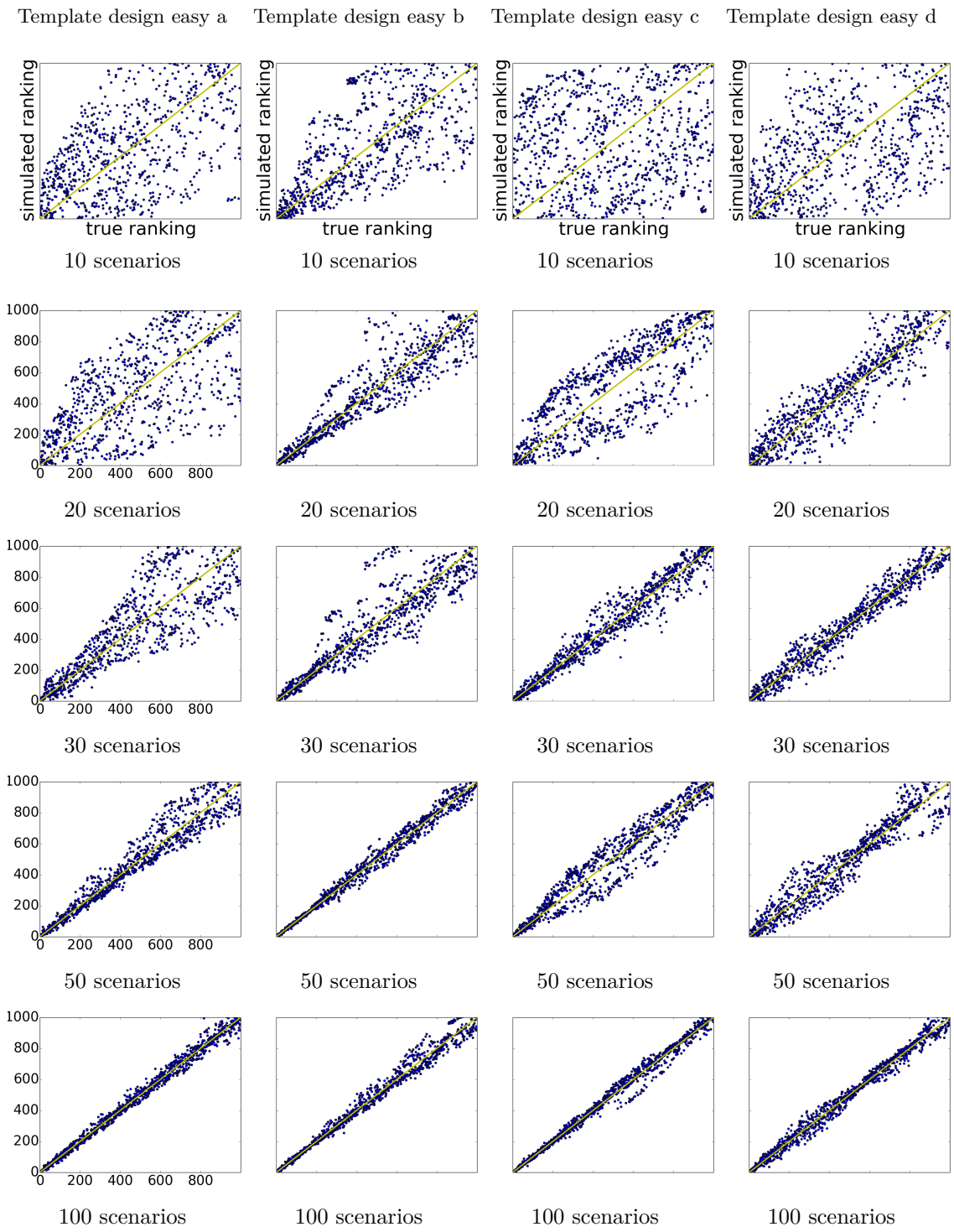


Figure 6.4: Ranking of candidates by objective value

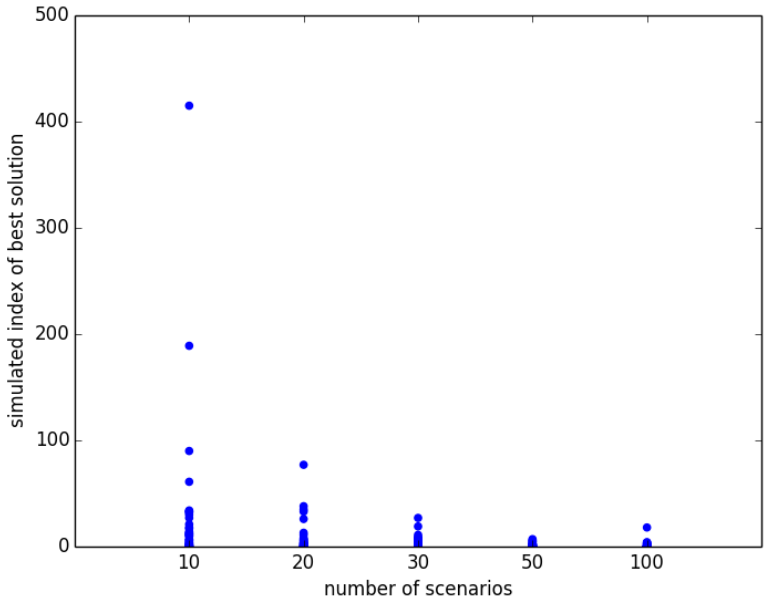


Figure 6.5: Simulated ranking of best candidate

the easy benchmark scales super linearly with respect to the number of scenarios. When solving a $\hat{\mathcal{P}}_{\mathcal{S}N}$ with $N = 2$ for the difficult benchmark an optimality gap of 32% remained on average after 3600 seconds. While the CP solver Chuffed outperforms CPLEX when solving a single scenario, Chuffed was also not able to solve any DE within the time out. The symmetry and dominance breaking constraints improved the performance by about 10% when solving a single scenario with Chuffed. Finally, to assess the time required to evaluate candidates we created models composed of 10 to 1000 scenarios with a fixed first stage and solved them using CPLEX. Clearly, evaluating a candidate is cheap, even for the three template benchmark.

	# scenarios	1	2	10	20	30	50	100	1000
Template design	generate Chuffed	0.03		-	-	-			
easy	generate CPLEX	2.9		367	1373	2925	-		
	evaluate			0.08	0.08	0.98	0.12	0.17	1.04
Template design	generate Chuffed[1]	3.8							
difficult	generate Chuffed[2]	4.2	-	-					
	generate CPLEX	77	-	-					
	evaluate			0.07	0.09	0.12	0.15	0.15	3.2

The symbol - indicates that the problem can not be solves in 3600 seconds

A blank cell indicates that no experiments where conducted

[1] Solved including symmetry and dominance breaking constraints

[2] Solved without symmetry and dominance breaking constraints

Table 6.2: Time to generate or evaluate a single candidate

In conclusion, our experiments have demonstrated that Simulation Based E&C is a valuable approach for solving stochastic problems with a large number of scenario, especially when solving an SAA is computationally challenging. As expected, the resulting optimality gap of Simulation Based E&C is not as tight as the gap obtained when using SAA.

6.4 Related work

Shapiro (2013) mentions that finding the precise origin of Monte Carlo sampling based methods to solve stochastic programs is difficult. He points out that the idea is rather natural and was discovered multiple times in different variations and contexts. In stochastic programming, early works can be attributed to Rubinstein and Shapiro (1990) and Robinson (1996), where this approach was called stochastic counterpart method and sample-path optimisation. In statistics the foundation for this type of Monte Carlo methods was laid by Geyer and Thompson (1992). In recent years however, most applications of the Monte Carlo based SAA method are based on works by Kleywegt et al. (2002) and Ahmed et al. (2002). We build upon their works, however our algorithm is different as it uses individual scenarios, instead of stochastic programs, to propose a large number of candidates rather than only few. In Constraint Programming, (Prestwich et al.; 2015) proposed a metaheuristic approach to solve SCOPs using standard filtering algorithms to handle hard constraints effectively. Our work is different as it does not focus on filtering algorithms, which are typically embedded in solvers and as such require modification in the solver itself, which we use as back-end.

Birattari et al. (2005) proposed a simulation based scheme to solve combinatorial optimisation problems that uses ant colony optimisation to propose candidates and racing

techniques for the evaluation. Racing has origins in machine learning (Maron; 1994) and has been widely applied in solver configuration (Birattari et al.; 2010). In a nutshell, racing techniques, e.g. the F-race (Birattari et al.; 2010), aim to minimise the number of evaluations required to reject a candidate. The candidates are sequentially evaluated over an increasing set of scenarios, until the low quality candidates can be rejected with confidence. Our ordering approach is different to racing as we reject candidates without statistical confidence, use instead an ordering approach, and therefore possibly lower the number of evaluation steps required.

6.5 Conclusion

We introduced Simulation Based E&C, an algorithm that scales well and is able solve problems that cannot be solved by the SAA method. To evaluate the performance of our algorithm, we compared it to the Monte Carlo based SAA method introduced by Kleywegt et al. (2002).

First, we demonstrated that the Monte Carlo based SAA method performs well on the stochastic template design problem, if one seeks to solve computationally easy instances. For more challenging problems, the SAA method fails to produce any results. Furthermore, we have demonstrated that the fundamental idea proposed by Ahmed (2013) remains true in the studied setting; the set of optimal scenario solutions is likely to contain a high quality solution for the true problem. To simulate the quality of candidates, we propose a three-stage algorithm. First, a moderate number of scenarios is used to reject all but the most promising candidates. Secondly, a large number of scenarios is used to determine the best few candidates. Lastly, a very large number of scenarios shall be used to reduce the variance of the objective value if required. We have shown experimentally that the evaluation scheme works in the proposed setting.

In conclusion we presented an algorithm that combines optimisation with simulation techniques to produce high quality policies. However, in contrast to the SAA procedure by Kleywegt et al. (2002) that requires solving sampled SCOPs to optimality, our algorithm scales to difficult problem instances where solving a SCOP is computationally challenging or simply not possible.

6.5.1 Limitations

Our experiments confirm that Sampling Based E&C is able to retrieve solutions of comparable quality to the Monte Carlo SAA method. However, in comparison, a relatively large optimality gap remains when using Simulation Based E&C. In conclusion, if the Monte Carlo SAA method is applicable, e.g. for problems where solving the SAA SCOP is trivial, it is likely to outperform our approach, due to the tight optimality gap . However, Simulation Based E&C scales well and is therefore well positioned to solve computationally challenging stochastic programs.

6.5.2 Future work

We introduced a promising algorithm that opens multiple research directions. First, it would be interesting to quantify the number of candidates required to have statistical confidence that a high quality candidate has been found. Likewise, determining the minimal number of scenarios required to simulate the candidate ranking with confidence would be beneficial, such that the parameters $N_1 \dots N_3$ and $M_1 \dots M_3$ can be determined automatically. Secondly, the presentation of the algorithm and the experiments of Simulation Based E&C where based on sampling scenarios randomly. Investigating the impact of introducing bias when sampling scenarios for candidate generation might be interesting as it could produce better results, without compromising the expected value (upper bound) simulation.

Chapter 7

Conclusions

The use of analytical methods to improve decision making is important to manage the ever increasing complexity of organisations, ensure that resources are used effectively and to progress in automating procedures. In many situations, decisions are implemented in environments that are governed by uncertainty and decision makers have to account for this uncertainty.

Stochastic programming is a common approach to solve combinatorial optimisation problems that are subject to uncertainty. Separating the modelling and solving of optimisation problems enabled traditional, deterministic optimisation its breakthrough and a similar separation is used to solve stochastic programs. Multiple modelling systems that cater for decision making under uncertainty have evolved over the years and share many aspects. All these systems are built on top of a modelling framework that was originally developed to model and solve deterministic optimisation problems. Thereafter, the expressiveness of the language was extended to allow for uncertainty models. Since solving a stochastic program as one single large deterministic equivalent model is notoriously hard, specialised back-end solvers have been developed.

This thesis proposed multiple improvements for existing frameworks to model and solve stochastic programs. While Stochastic MiniZinc has been used throughout the thesis to illustrate how our novel concepts can be applied in practice, all contributions are equally applicable to be integrated in comparable systems. First, we have shown how to create better stochastic programming models. Secondly, we developed algorithms to solve combinatorial stochastic programs using decomposition methods and off-the-self solvers. And lastly, we proposed a method that combines E&C with simulation. This chapter presents the conclusions and addresses the contributions.

7.1 Modelling

Chapter 3 looked at improving stochastic programming formulations using modelling techniques that are common practice in constraint programming.

Firstly, we addressed the role of symmetry and dominance breaking constraints in the context of stochastic programming. We highlighted problems that occur if symmetry and dominance breaking constraints are used in the decision model without explicitly

marking them as auxiliary. Thereafter, we showed how to compile symmetry and dominance breaking constraints correctly when compiling a decision model into a stochastic program. In conclusion, using auxiliary constraints to break symmetries and dominance relations improves the decision model, however unless the modelling frameworks provides the appropriate support, auxiliary constraints must be used with care.

Secondly, we proposed a compilation technique to strengthen the subproblems for scenario decomposition based algorithms. The goal of this technique is to reduce the number of infeasible solutions explored during the search by lifting appropriate constraints from being local to a specific scenario to act in every subproblem. We compared our approach conceptually to scenario bundling, a well known technique to improve the scenario decomposition. Finally, we used an empirical study to showcase the positive effects of lifting constraints.

This chapter has shown how to compile symmetry and dominance breaking constraints correctly. However, the additional functionality has not yet been implemented in the Stochastic MiniZinc compiler, we considered this future work. Furthermore, we proposed the idea of lifting constraints to strengthen the scenario subproblems and demonstrated its effect using a facility location problem with capacity and other side constraints. However, further experiments are required to determine contexts in which lifting a constraint is beneficial. Thereafter, a design decision is required; should appropriate constraints be automatically lifted or is it better to let the modeller annotate appropriate constraints explicitly?

7.1.1 Evaluate and Cut with Diving

Chapter 4 looked at improving an algorithm to solve two-stage stochastic problems with binary first-stage variables introduce by Ahmed (2013) that we call Evaluate and Cut (E&C). The algorithm is based on the scenario decomposition, composed of three steps and utilises off-the-shelf solvers. Firstly, solving each scenario yields a set of candidates. A candidate is a first-stage variable assignment that matches the optimal solution of a scenario. Thereafter, the candidates are evaluated across all scenarios to assess their quality with respect to the stochastic problem. And lastly, by adding a nogood constraint to each scenario the candidates are excluded from the search in subsequent iterations. Repeating the three steps is guaranteed to retrieve the optimal solution. The E&C algorithm can be applied to solve stochastic combinatorial optimisation problems that are formalised in a high-level modelling language. The algorithm performs strongly in finding the optimal solution quickly, however, depending on the problem class requires a long time to prove optimality. We proposed two techniques to improve E&C, demonstrated empirically their performance and published the benchmark we used for the experiments.

Firstly, we have shown how to use E&C in combination with constraint programming solvers to tackle problems with integer variables in the first stage. We also introduced *vertical learning*, a technique to solve subproblems faster by reusing information that was learnt in previous iterations. Lacy Clause Generation, or learning solvers, are a type of CP solver that explains why a decision led to failure and constructs constraints that prevent

the solver from repeating the same decision later during the search. In vertical learning these learnt constraints, are reused for solving subproblems in subsequent iterations faster.

Secondly, we introduced *diving*, a method aimed at proving optimality faster when using E&C to solve stochastic programs. The diving procedure is able to retrieve nogood constraints that prune multiple candidates at once without compromising the completeness of E&C. We created, used and published a two-stage generalised assignment problem to evaluate the effects of vertical learning and diving.

Vertical learning has shown to be effective, however further analysis into how to effectively utilise learnt constraints could yield additional improvements. Furthermore, the diving method uses a variable selection heuristic that impacts the performance of diving greatly and exploring the impact of alternative selection heuristics would be interesting.

7.1.2 Recursive Evaluate and Cut

Chapter 5 looked at generalising the E&C (Ahmed; 2013) algorithm to multi-stage problems.

We have shown how the E&C algorithm can be used to recursively solve combinatorial multi-stage problems. Firstly, we illustrated how to implement a naïve recursive version of E&C and highlighted its weaknesses. Thereafter, we developed an alternative implementation and evaluated its performance empirically using a four-stage facility location problem with capacity and other side constraints. As in standard E&C, the multi-stage version utilises off-the-shelf solvers and is therefore well positioned as back-end solver for stochastic programming modelling frameworks, such as Stochastic MiniZinc.

Evaluate and Cut and its recursive version are limited to solve problems with discrete variables. The algorithm is complete as only a finite number of candidates with discrete variables can be retrieved during the search, a property that does not hold true for continuous variables. It would be interesting to develop a version of E&C that is not restricted to discrete variables. Essentially, the challenge is to find ways of creating nogood constraints that are only composed of discrete variables.

7.1.3 Simulation Based Evaluate and Cut

Chapter 6 looked at solving stochastic problems that cannot be solved optimally, either because of their size (e.g. too many scenarios) or because the scenario-based uncertainty model is only an approximation of the random variables.

We introduced *Simulation Based Evaluate and Cut*, an algorithm that combines optimisation with simulation. As in standard E&C, candidates are generated by solving individual scenario subproblems. The novel idea is to then evaluate the candidates using a Monte Carlo based simulation approach. We demonstrated the effectiveness of our algorithm using a two-stage template design problem with uncertain demand and compared it to the state of the art technique to solve prohibitively large stochastic programs. In conclusion, on easy instances Simulation Based E&C produced solutions of comparable quality, however our algorithm scales much better compared to the other approach and is

therefore able to solve computationally challenging instances that could not be solved otherwise. The state of the art technique requires solving multiple stochastic programs and is therefore restricted by the complexity of those stochastic problems. In contrast, Simulation Based E&C only requires solving scenarios, essentially deterministic optimisation problems, which is computationally much less expensive.

We presented an unbiased version of simulation based E&C. In other words, scenarios were randomly selected to generate and to evaluate candidates. However, introducing bias into the sampling is an interesting future research direction. Firstly, to generate candidates; so far, we randomly sampled scenarios to obtain one candidate per scenario. Introducing a measure that ranks scenarios by how likely they are to return high quality solutions might improve the search. Secondly, to evaluate the quality of candidates; in our experiments we evaluated the quality of a candidate over a randomly sampled set of scenarios. A more representative set of samples, for example by using the xor-sampling techniques as proposed by Gomes et al. (2007), may be able to reduce the number of samples required to achieve the same evaluation quality.

7.1.4 Concluding Remarks

This thesis contributes to decision making under uncertainty by improving on existing technologies and combining ideas from constraint programming and stochastic programming. Firstly, we have shown that providing functionality that enables a modeller to express strong models is an important aspect with regard to improved solving performance. Secondly, we developed algorithms and extensions thereof that utilise off-the-self solvers to effectively tackle stochastic programs. We focussed on algorithms that use existing solver technology because of the following reasons. Firstly, by choosing appropriate technology to solve the subproblems, the algorithms show strong performance on a variety of problem classes. Secondly, improving off-the-self solver performance is an active field of research, and our algorithm benefit automatically as the performance of these solvers increases.

The focus of this thesis is on how to effectively solve stochastic programs. To continue our line of research, it would be interesting to develop methods that communicate the impact of a policy effectively. To implement a policy in practice, the decision maker requires a solid understanding of how the proposed actions affect their operations. This could possibly be achieved using visualisation or interactive optimisation, where the decision maker is able to interfere with the search.

In conclusion, this thesis presents contributions at the intersection of stochastic and constraint programming. We addressed both aspects of a modelling system: firstly, modelling and secondly solving a stochastic problem. We are convinced that our work has shown that solving stochastic programs using high-level modelling systems is promising.

Appendix A

Benchmark Models

A.1 Template Design

The deterministic version of the template design model was originally published by Proll and Smith (1998) and a MiniZinc model including instances can be found on the CSPLib (Gent and Walsh; 1999).

A.1.1 Template Design in Stochastic MiniZinc

The following model is based on the original model by Proll and Smith (1998) and annotated with Stochastic MiniZinc instructions.

```
% Template design
% Problem 002 in CSPLib
%-----------------------------------------------------------------------%
% Based on "ILP and Constraint Programming Approaches to a Template
% Design Problem", Les Proll and Barbara Smith, School of Computing
% Research Report 97.16, University of Leeds, May 1997.
%-----------------------------------------------------------------------%

include "globals.mzn";
include "stochastic.mzn"

int: S;                  % Number of slots per template.
int: t;                  % Number of templates.
int: n;                  % Number of variations.
% How much of each variation we must print?
array[1..n] of int: d :: stage(2);

% Lower and upper bounds for the total production.
int: llower = ceil(sum(i in 1..n)(int2float(d[i]))/int2float(S))::stage(2);
int: lupper = 2*llower :: stage(2);

% # Slots allocated to variation i in template j
array[1..n,1..t] of var 0..S: p :: stage(1);

% # Pressings of template j.
array[1..t] of var 1..lupper: R :: stage(2);
```

```
27
28 % Sum of all Rj.
29 var llower..lupper: Production ::stage(2);
30
31 % Production x S - sum(d[i])
32 var 0..lupper-llower: Surplus;
33
34 % First, set up Production to be the sum of the Rj
35 constraint Production = sum(i in 1..t)(R[i]);
36
37 % The number of slots occupied in each template is S.
38 constraint forall(j in 1..t)(sum(i in 1..n)(p[i,j]) = S);
39
40 % Enough of each variation is printed.
41 constraint forall(i in 1..n)(sum(j in 1..t)(p[i,j]*R[j]) >= d[i]);
42
43 %% Symmetry breaking constraints
44 % Variations with the same demand are symmetric.
45 constraint symmetry_breaking_constraint(
46              forall(i in 1..n-1) (
47               d[i] == d[i+1] ->
48               lex_lesseq([p[i,  j] | j in 1..t],
49               [p[i+1,j] | j in 1..t])
50              )
51              );
52
53 %% Dominance breaking constraints
54 constraint dominance_breaking_constraint(
55             forall(i in 1..n-1) (
56              d[i] < d[i+1] ->
57              sum (j in 1..t) (p[i,j]*R[j])
58              <= sum (j in 1..t) (p[i+1,j]*R[j])
59             )
60           );
61
62 % Set up surplus, which is bounded as production is bounded.
63 constraint Surplus = Production*S - sum(i in 1..n)(d[i]);
64
65 % The surplus of each variation is also limited by the surplus.
66 constraint forall(k in 1..n)(sum(j in 1..t)(p[k,j]*R[j]-d[k]) <= Surplus);
67
68 % The surplus of the first k variations is limited by the surplus.
69 constraint forall(k in 2..n-1)
70             (sum(j in 1..t, m in 1..k)( p[m,j]*R[j]-d[m] ) <= Surplus);
71
72 % Minimize the production.
73 solve :: int_search(array1d(1..n*t,p) ++ R,
74             input_order, indomain_min, complete)
75             minimize Production :: expected;
```

A.1.2 Template Design Deterministic Equivalent

The following model contains a deterministic equivalent formulation for the template design model.

```
% Two-stage template design deterministic equivalet
int: S;                 % Number of slots per template.
int: t;                 % Number of templates.
int: n;                 % Number of variations.
array[SCENARIOS,1..n] of int: d; % Scenario dependent Demand

int: nb_scen;
set of int: SCENARIOS = 1..nb_scen;

% # Slots allocated to variation i in template j
array[1..n,1..t] of var 0..S: p;

% # Pressings of template j.
array[SCENARIOS, 1..t] of var intr: R;
array[SCENARIOS] of var int: Production;
array[SCENARIOS] of var int: Surplus;

% First, set up Production to be the sum of the Rj in each scenario
constraint forall(sc in SCENARIOS)(
             Production[sc] = sum(i in 1..t)(R[sc,i]));


% The number of slots occupied in each template is S.
constraint forall(j in 1..t)(sum(i in 1..n)(p[i,j]) = S);

% Enough of each variation is printed.
constraint forall(sc in SCENARIOS)(
             forall(i in 1..n)
              (sum(j in 1..t)(p[i,j]*R[sc,j]) >= d[sc,i]));

% Set up surplus, which is bounded as production is bounded.
constraint forall(sc in SCENARIOS)(
             Surplus[sc] = Production[sc] * S - sum(i in 1..n)(d[sc,i]));

% The surplus of each variation is also limited by the surplus.
constraint forall(sc in SCENARIOS)(
             forall(k in 1..n)
              (sum(j in 1..t)(p[k,j]*R[sc,j]-d[sc,k]) <= Surplus[sc]));

% The surplus of the first k variations is limited by the surplus.
constraint forall(sc in SCENARIOS)(
             forall(k in 2..n-1)
              (sum(j in 1..t, m in 1..k)( p[m,j]*R[sc,j]-d[sc,m] )
               <= Surplus[sc]));

constraint expectedProduction = sum(sc in SCENARIOS)
             (Production[sc])/nb_scen;

```

```
% Minimize the production.
solve :: int_search(array1d(1..n*t,p) ++ array1d(1..nb_scen*t,R),
             input_order, indomain_min, complete)
              minimize expectedProduction;
```

A.1.3 Template Design Data

For each package (n) four or five random numbers where drawn. To get the scenarios, we considered all possible combinations. The instruction *all_possible_combinations_of* is not actual MiniZinc syntax, but rather used to describe that we consider all possible combinations as the complete set of scenarios.

```
%template_design_4069_t2_a
%number of slots on each template
S = 9;
%number of templates
t = 2;
%number of different packages
n = 6;
%number of scenarios
nb_scen = 4096;
%demand
d = all_possible_combinations_of(
[|465,960,955, 960,865,480|
  490,335,380, 560,375,675|
  820,820,365, 650,990,560|
 1015,295,270,1060,855,565|]);
```

```
%template_design_4069_t2_b
%number of slots on each template
S = 9;
%number of templates
t = 2;
%number of different packages
n = 6;
%number of scenarios
nb_scen = 4096;
%demand
d = all_possible_combinations_of(
[|1040,800,660,520,315,480|
   425,265,710,465,990,850|
   635,795,550,970,750,895|
   755,600,815,345,990,615|]);
```

```
%template_design_4069_t2_c
%number of slots on each template
S = 9;
%number of templates
t = 2;
%number of different packages
n = 6;
```

```
%number of scenarios
nb_scen = 4096;
%demand
d = all_possible_combinations_of(
[|640,320,665,290, 810,1000|
  600,345,530,355, 940, 280|
  885,315,790,870, 885, 930|
  990,820,625,290,1005, 955|]);
```

```
%template_design_4069_t2_d
%number of slots on each template
S = 9;
%number of templates
t = 2;
%number of different packages
n = 6;
%number of scenarios
nb_scen = 4096;
%demand
d = all_possible_combinations_of(
[| 320,785,440,720,785,745|
   465,970,385,920,810,860|
  1075,485,995,445,995,990|
   755,830,615,775,740,265|]);
```

```
%template_design_3125_t3_a
%number of slots on each template
S = 9;
%number of templates
t = 3;
%number of different packages
n = 5;
%number of scenarios
nb_scen = 3125;
%demand
d = all_possible_combinations_of(
[|985, 530,520, 880,745|
  795, 595,890, 955,490|
  660, 900,635, 590,660|
  470,1070,675,1070,765|
  755, 790,750, 935,615|]);
```

```
%template_design_3125_t3_a
%number of slots on each template
S = 9;
%number of templates
t = 3;
%number of different packages
n = 5;
%number of scenarios
nb_scen = 3125;
%demand
```

```
d = all_possible_combinations_of(
[|385, 800, 665, 865,480
  450,1025, 275,1010,640|
  540, 890, 940, 315,875|
  890, 400,1070, 845,300|
  655, 625, 935, 495,305|]);
```

A.1.4 Stochastic Assignment and Scheduling Problem

```
% Including files
include "globals.mzn";
% Parameters
int: no_mach;   % Number of machines
int: no_jobs;   % Number of jobs
int: no_task;   % Number of total tasks
int: no_optt;   % Number of total optional tasks

set of int: Mach  = 1..no_mach;
set of int: Jobs  = 1..no_jobs;
set of int: Tasks = 1..no_task;
set of int: OptTs = 1..no_optt;

array [Jobs] of set of int: tasks;
array [Tasks] of set of int: optts;

array [OptTs] of int: optt_mach;
array [SCENARIOS1,OptTs] of int: optt_dur;


array [Jobs] of int: last_task = [ max(tasks[j]) | j in Jobs ];
%---------implications for multi scenarion solving ---------------
int: nbScenarios;
set of int: SCENARIOS1 = 1..nbScenarios;
int: first_scen;
int: last_scen;
set of int: SCENARIOS = first_scen..last_scen;
array[SCENARIOS1] of int: weights;

%-------end of multi scenario addons ----------------
array [Tasks] of int: task_job =
[ min(j in Jobs where t in tasks[j])(j) | t in Tasks ];
array [SCENARIOS,Tasks] of int: task_mins =
array2d(SCENARIOS,Tasks,[ sum(k in tasks[task_job[t]])(if k < t
       then task_mind[s,k] else 0 endif)
|   s in SCENARIOS, t in Tasks ]);
array [SCENARIOS,Tasks] of int: task_maxs =
array2d(SCENARIOS,Tasks,[ t_max[s] -
sum(k in tasks[task_job[t]])(if k < t then 0 else task_mind[s,k] endif)
|   s in SCENARIOS,  t in Tasks ]);

array [SCENARIOS,Tasks] of int: task_mind =
```

```
43 array2d(SCENARIOS,Tasks,[ min(o in optts[t])(optt_dur[s,o])
44 | s in SCENARIOS,t in Tasks ]);
45
46 array [SCENARIOS,Tasks] of int: task_maxd =
47 array2d(SCENARIOS,Tasks,[ max(o in optts[t])(optt_dur[s,o])
48 | s in SCENARIOS, t in Tasks ]);
49
50 % Additional deirved parameters for optional tasks
51 %
52 array [OptTs] of int: optt_task =
53 [ min(t in Tasks where o in optts[t])(t) | o in OptTs ];
54
55 array[SCENARIOS1] of int: min_dur = [ min([optt_dur[s,t] | t in OptTs])
56 | s in SCENARIOS1];
57 array[SCENARIOS1] of int: max_dur = [ max([optt_dur[s,t] | t in OptTs])
58 | s in SCENARIOS1];
59 set of int: Durs = min(min_dur)..max(max_dur);
60
61 % Parameters related to the planning horizon
62 %
63 array[SCENARIOS1] of int: t_max = [sum(t in Tasks)(max(o in optts[t])
64 (optt_dur[s,o])) | s in SCENARIOS1];
65
66 set of int: Times = 0..max(t_max);
67 % Variables
68
69 % Start time variables for tasks
70 %
71 array [SCENARIOS,Tasks] of var Times: start =
72 array2d(SCENARIOS,Tasks,[ let { var task_mins[s,t]..task_maxs[s,t]: k }
73 in k | s in SCENARIOS, t in Tasks ]);
74
75 % Duration variables for tasks
76 %
77 array [SCENARIOS,Tasks] of var Durs: dur =
78 array2d(SCENARIOS,Tasks,[ if task_mind[s,t] =
79 task_maxd[s,t] then task_mind[s,t] else
80 let { var task_mind[s,t]..task_maxd[s,t]: d } in d endif
81 |   s in SCENARIOS,t in Tasks ]);
82
83 % Variables whether an optional task is executed
84 %
85 array [OptTs] of var bool: b;
86
87 array[SCENARIOS] of var Times: de_objective;
88
89 set of int: StochTimes = 0..sum(t_max);
90 var StochTimes: objective;
91 % Constraints
92
93 % Precedence relations
94 %
```

```
95  constraint
96  forall(s in SCENARIOS)(
97  forall(j in Jobs, i in tasks[j] where i < last_task[j])(
98  start[s,i] + dur[s,i] <= start[s,i + 1]
99  )
100 );
101
102 % Duration constraints
103 %
104 constraint
105 forall(o in OptTs,s in SCENARIOS)(
106 let { int: t = optt_task[o] } in (
107 if card(optts[t]) = 1 then
108 b[o] = true
109 else
110 b[o] -> dur[s,t] = optt_dur[s,o]
111 endif
112 )
113 );
114
115 % Optional tasks' constraints
116 %
117 constraint
118 forall(t in Tasks where card(optts[t]) > 1)(
119 ( sum(o in optts[t])(bool2int(b[o])) <= 1     )
120 /\  ( exists(o in optts[t])(b[o])                 )
121 );
122
123 constraint
124 forall(t in Tasks where card(optts[t]) = 2)(
125 let {
126 int: o1 = min(optts[t]),
127 int: o2 = max(optts[t])
128 } in ( b[o1] <-> not(b[o2]) )
129 );
130
131 % Resource constraints
132 %
133 constraint
134 forall(m in Mach,s in SCENARIOS)(
135 let {
136 set of int: MTasks = { o | o in OptTs where optt_mach[o] = m }
137 } in (
138 cumulative(
139 [ start[s,optt_task[o]] | o in MTasks ],
140 [ optt_dur[s,o]         | o in MTasks ],
141 [ bool2int(b[o])      | o in MTasks ],1)));
142
143 constraint forall(s in SCENARIOS)(
144 forall(j in Jobs)(start[s,last_task[j]] + dur[s,last_task[j]]
145 <= de_objective[s]));
146
```

```
147 constraint objective = sum(s in SCENARIOS)(weights[s]*de_objective[s]);
148 % Solve item
149 solve minimize objective;
```

A.2 Four-stage stochastic facility location problem

The following model contains a deterministic equivalent model formulation of the four-stage stochastic facility location problem.

```
1  int: nbFacility;
2  int: nbCustomers;
3  int: nbStages;
4  int: nb1Scen;
5  int: nb2Scen;
6  int: nb3Scen;
7  int: relocationPenalty;
8
9  set of int: FACILITIES = 1..nbFacility;
10 set of int: CUSTOMERS = 1..nbCustomers;
11 set of int: STAGES = 1..nbStages;
12 set of int: SCENARIOS1 = 1..nb1Scen;
13 set of int: SCENARIOS2 = 1..nb1Scen*nb2Scen;
14 set of int: SCENARIOS3 = 1..nb1Scen*nb2Scen*nb3Scen;
15
16 array[FACILITIES,CUSTOMERS] of int: distance;
17 array[1..nbStages-1,FACILITIES] of int: setupCost;
18 array[FACILITIES] of int: capacity;
19 array[CUSTOMERS] of int: demand;
20 array[SCENARIOS1,CUSTOMERS] of 0..1: c2_availability;
21 array[SCENARIOS2,CUSTOMERS] of 0..1: c3_availability;
22 array[SCENARIOS3,CUSTOMERS] of 0..1: c4_availability;
23
24 %vaiables
25 array[FACILITIES] of var 0..1: fl1;
26 array[SCENARIOS1,FACILITIES] of var 0..2: fl2;
27 array[SCENARIOS2,FACILITIES] of var 0..3: x;
28
29 array[SCENARIOS1,CUSTOMERS] of var 0..nbFacility: c2f2;
30 array[SCENARIOS2,CUSTOMERS] of var 0..nbFacility: c2f3;
31 array[SCENARIOS3,CUSTOMERS] of var 0..nbFacility: c2f4;
32
33 array[STAGES] of var 0.0..1000000.0: transportCost;
34 array[STAGES] of var 0.0..1000000.0: warehouseCost;
35 array[STAGES] of var 0.0..1000000.0: relocationCost;
36
37 var float: obj;
38
39 %auxiliary variables
40 array[SCENARIOS1,FACILITIES] of var 0..10000: aggregateDemand2;
41 array[SCENARIOS2,FACILITIES] of var 0..10000: aggregateDemand3;
42 array[SCENARIOS3,FACILITIES] of var 0..10000: aggregateDemand4;
```

```
43
44 %warehouse setup cost
45 constraint warehouseCost[1] = sum(f in FACILITIES)
46             (fl1[f] * setupCost[1,f]);
47 constraint warehouseCost[2] = sum(s1 in SCENARIOS1,f in FACILITIES)
48             (bool2int(fl2[s1,f] == 2) * setupCost[2,f]) / (nb1Scen);
49 constraint warehouseCost[3] = sum(s2 in SCENARIOS2,f in FACILITIES)
50             (bool2int(x[s2,f] == 3) * setupCost[3,f]) / (nb1Scen*nb2Scen);
51 constraint warehouseCost[4] = 0;
52
53 %warehouse consistency
54 constraint forall(s1 in SCENARIOS1, f in FACILITIES)(
55             fl1[f] = 1 <-> fl2[s1,f] = 1
56            );
57 constraint forall(s2 in SCENARIOS2, f in FACILITIES)(
58             fl1[f] = 1 <-> x[s2,f] = 1
59            );
60 constraint forall(s1 in SCENARIOS1, s2 in 1..nb2Scen, f in FACILITIES)(
61             fl2[s1,f] = 2 <-> x[(s1-1)*nb2Scen + s2,f] = 2
62            );
63
64 %transport cost
65 constraint transportCost[1] = 0;
66 constraint transportCost[2] = sum(s1 in SCENARIOS1, c in CUSTOMERS)(
67             c2_availability[s1,c] * distance[c2f2[s1,c],c]) / nb1Scen;
68 %third transport cost
69 constraint transportCost[3] = sum(s2 in SCENARIOS2, c in CUSTOMERS)(
70             c3_availability[s2,c] * distance[c2f3[s2,c],c])
71             / (nb1Scen * nb2Scen);
72 constraint transportCost[4] = sum(s3 in SCENARIOS3, c in CUSTOMERS)(
73             c4_availability[s3,c] * distance[c2f4[s3,c],c])
74             / (nb1Scen * nb2Scen * nb3Scen);
75
76 %total demand for each facility
77 constraint forall(s1 in SCENARIOS1, f in FACILITIES)(
78             aggregateDemand2[s1,f] = sum(c in CUSTOMERS)(
79             c2_availability[s1,c] * bool2int(c2f2[s1,c] == f) * demand[c])
80            );
81 constraint forall(s2 in SCENARIOS2, f in FACILITIES)(
82             aggregateDemand3[s2,f] = sum(c in CUSTOMERS)(
83             c3_availability[s2,c] * bool2int(c2f3[s2,c] == f) * demand[c])
84            );
85 constraint forall(s1 in SCENARIOS1, s2 in 1..nb2Scen, f in FACILITIES)(
86             aggregateDemand3[(s1-1)*nb2Scen + s2 ,f] > 0
87             -> fl2[s1,f] > 0 /\ fl2[s1,f] < 3
88 );
89
90 %Auxiliary constraints to sum up the total demand per facility
91 constraint forall(s2 in SCENARIOS2, f in FACILITIES)(
92             aggregateDemand3[s2,f] = sum(c in CUSTOMERS)(
93             c3_availability[s2,c] * bool2int(c2f3[s2,c] == f) * demand[c]));
94 constraint forall(s2 in SCENARIOS2, f in FACILITIES)(
```

```
95              aggregateDemand3[s2,f] = sum(c in CUSTOMERS)(
96              c3_availability[s2,c] * bool2int(c2f3[s2,c] == f) * demand[c]));
97  constraint forall(s3 in SCENARIOS3, f in FACILITIES)(
98              aggregateDemand4[s3,f] = sum(c in CUSTOMERS)(
99              c4_availability[s3,c] * bool2int(c2f4[s3,c] == f) * demand[c]));
100
101 %Capacity constraint on each of the facilities in each stage
102 constraint forall(s1 in SCENARIOS1,f in FACILITIES)(
103             aggregateDemand2[s1,f] <= capacity[f]);
104 constraint forall(s2 in SCENARIOS2,f in FACILITIES)(
105             aggregateDemand3[s2,f] <= capacity[f]);
106 constraint forall(s3 in SCENARIOS3,f in FACILITIES)(
107             aggregateDemand4[s3,f] <= capacity[f]);
108
109 %ensure that a facility is open if demand is assigned to it
110 constraint forall(s1 in SCENARIOS1, f in FACILITIES)(
111             aggregateDemand2[s1,f] > 0 -> fl1[f] = 1);
112 constraint forall(s1 in SCENARIOS1, s2 in 1..nb2Scen, f in FACILITIES)(
113             aggregateDemand3[(s1-1)*nb2Scen + s2 ,f] > 0
114             -> fl2[s1,f] > 0 /\ fl2[s1,f] < 3);
115 constraint forall(s2 in SCENARIOS2, s3 in 1..nb3Scen, f in FACILITIES)(
116             aggregateDemand4[(s2-1)*nb3Scen + s3 ,f] > 0
117             -> x[s2,f] != 0);
118
119 %relocation cost
120 constraint relocationCost[1] = 0;
121 constraint relocationCost[nbStages] = 0;
122 constraint relocationCost[2] = sum(s1 in SCENARIOS1, c in CUSTOMERS)(
123             sum(s2 in 1..nb2Scen)(c2_availability[s1,c] *
124             bool2int(c2f2[s1,c] !=  c2f3[(s1-1)*nb2Scen + s2,c]))
125             ) / (nb1Scen * nb2Scen) * relocationPenalty;
126 constraint relocationCost[3] = sum(s2 in SCENARIOS2, c in CUSTOMERS)(
127             sum(s3 in 1..nb3Scen)(c3_availability[s2,c] *
128             bool2int(c2f3[s2,c] !=  c2f4[(s2-1)*nb3Scen + s3,c]))
129             ) / (nb1Scen * nb2Scen * nb3Scen) * relocationPenalty;
130
131 constraint obj = sum(i in STAGES)(warehouseCost[i]
132             + transportCost[i] + relocationCost[i]);
133
134 solve minimize obj;
```

References

Ahmed, S. (2013). A scenario decomposition algorithm for 0–1 stochastic programs, *Operations Research Letters* **41**(6): 565–569.

Ahmed, S., King, A. J. and Parija, G. (2003). A multi-stage stochastic integer programming approach for capacity expansion under uncertainty, *Journal of Global Optimization* **26**(1): 3–24.

Ahmed, S., Luedtke, J., Song, Y. and Xie, W. (2017). Nonanticipative duality, relaxations, and formulations for chance-constrained stochastic programs, *Mathematical Programming* **162**(1-2): 51–81.

Ahmed, S., Shapiro, A. and Shapiro, E. (2002). The sample average approximation method for stochastic programs with integer recourse, *Submitted for publication* pp. 1–24.

Albareda-Sambola, M., Van Der Vlerk, M. H. and Fernández, E. (2006). Exact solutions to a class of stochastic generalized assignment problems, *European journal of operational research* **173**(2): 465–487.

Aldasoro, U., Escudero, L. F., Merino, M. and Pérez, G. (2017). A parallel branch-and-fix coordination based matheuristic algorithm for solving large sized multistage stochastic mixed 0–1 problems, *European Journal of Operational Research* **258**(2): 590–606.

Alonso-Ayuso, A., Escudero, L. F. and Ortuno, M. T. (2003). Bfc, a branch-and-fix coordination algorithmic framework for solving some types of stochastic pure and mixed 0–1 programs, *European Journal of Operational Research* **151**(3): 503–519.

Applegate, D. L., Bixby, R. E., Chvatal, V. and Cook, W. J. (2006). *The traveling salesman problem: a computational study*, Princeton university press.

Apt, K. R. and Wallace, M. (2006). *Constraint logic programming using ECLiPSe*, Cambridge University Press.

Arabani, A. B. and Farahani, R. Z. (2012). Facility location dynamics: An overview of classifications and applications, *Computers & Industrial Engineering* **62**(1): 408–420.

Babaki, B., Guns, T. and De Raedt, L. (2017). Stochastic constraint programming with and-or branch-and-bound, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 539–545.

Barták, R. (1999). Constraint programming: In pursuit of the holy grail, *Proceedings of the Week of Doctoral Students (WDS99)*, pp. 555–564.

Basçiftci, B., Ahmed, S., Gebraeel, N. and Yildirim, M. (2017). Integrated generator maintenance and operations scheduling under uncertain failure times.

Beldiceanu, N., Carlsson, M., Demassey, S. and Petit, T. (2007). Global constraint catalogue: Past, present and future, *Constraints* **12**(1): 21–62.

Bezanson, J., Edelman, A., Karpinski, S. and Shah, V. B. (2017). Julia: A fresh approach to numerical computing, *SIAM review* **59**(1): 65–98.

Birattari, M., Balaprakash, P. and Dorigo, M. (2005). Aco/f-race: Ant colony optimization and racing techniques for combinatorial optimization under uncertainty, *MIC 2005: The 6th Metaheuristics International Conference*, Vienna, Austria: University of Vienna, Department of Business Administration, pp. 107–112.

Birattari, M., Yuan, Z., Balaprakash, P. and Stützle, T. (2010). F-race and iterated f-race: An overview, *Experimental methods for the analysis of optimization algorithms*, Springer, pp. 311–336.

Birge, J. R. and Louveaux, F. (2011). *Introduction to stochastic programming*, Springer Science & Business Media.

Bisschop, J. and Roelofs, M. (2006a). Aimms language reference, Lulu.com.

Bisschop, J. and Roelofs, M. (2006b). Aimms language reference, Lulu.com, chapter 19, pp. 307–326.

Bixby, R. E. (2012). A brief history of linear and mixed-integer programming computation, *Documenta Mathematica* pp. 107–121.

Björdal, G., Monette, J.-N., Flener, P. and Pearson, J. (2015). A constraint-based local search backend for minizinc, *Constraints* **20**(3): 325–345.

Boland, N., Christiansen, J., Dandurand, B., Eberhard, A., Linderoth, J., Luedtke, J. and Oliveira, F. (2018). Combining progressive hedging with a frank–wolfe method to compute lagrangian dual bounds in stochastic mixed-integer programming, *SIAM Journal on Optimization* **28**(2): 1312–1336.

Brandimarte, P. (1993). Routing and scheduling in a flexible job shop by tabu search, *Annals of Operations research* **41**(3): 157–183.

Bussieck, M. R. and Meeraus, A. (2004). General algebraic modeling system (gams), *Modeling languages in mathematical optimization*, Springer, pp. 137–157.

Carøe, C. C. and Schultz, R. (1999). Dual decomposition in stochastic integer programming, *Operations Research Letters* **24**(1-2): 37–45.

Chu, G., De La Banda, M. G., Mears, C. and Stuckey, P. J. (2014). Symmetries, almost symmetries, and lazy clause generation, *Constraints* **19**(4): 434–462.

Chu, G. G. (2011). Improving combinatorial optimization.

Chu, G. and Stuckey, P. J. (2012). Inter-instance nogood learning in constraint programming, *Principles and Practice of Constraint Programming*, Springer, pp. 238–247.

Chu, G. and Stuckey, P. J. (2015). Dominance breaking constraints, *Constraints* **20**(2): 155–182.

Chu, G. and Stuckey, P. J. (2016). Symmetry declarations for minizinc.

Cohen, D., Jeavons, P., Jefferson, C., Petrie, K. E. and Smith, B. M. (2006). Symmetry definitions for constraint satisfaction problems, *Constraints* **11**(2-3): 115–137.

Colombani, Y. and Heipcke, S. (2002). Mosel: an extensible environment for modeling and programming solutions, *Proceedings of CP-AI-OR*, Vol. 2, pp. 277–290.

Colquhoun, D. (1971). *Lectures on biostatistics: an introduction to statistics with applications in biology and medicine*, David Colquhoun.

Crainic, T. G., Hewitt, M. and Rei, W. (2014). Scenario grouping in a progressive hedging-based meta-heuristic for stochastic network design, *Computers & Operations Research* **43**: 90–99.

Dantzig, G. B. and Infanger, G. (1991). Large-scale stochastic linear programs: Importance sampling and benders decomposition, *Technical report*, STANFORD UNIV CA SYSTEMS OPTIMIZATION LAB.

de la Banda, M. G., Marriott, K., Rafeh, R. and Wallace, M. (2006). The modelling language zinc, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 700–705.

Deng, Y., Ahmed, S., Lee, J. and Shen, S. (2017). Scenario grouping and decomposition algorithms for chance-constrained programs, *Available on Optimization Online http://www. optimizationonline. org/DB HTML/2017/02/5853. html* .

Dunning, I., Huchette, J. and Lubin, M. (2017). Jump: A modeling language for mathematical optimization, *SIAM Review* **59**(2): 295–320.

Dupačová, J., Gröwe-Kuska, N. and Römisch, W. (2003). Scenario reduction in stochastic programming, *Mathematical programming* **95**(3): 493–511.

Ellison, F., Mitra, G. and Zverovich, V. (2010). Fortsp: a stochastic programming solver, *OptiRisk Systems* .

Escudero, L. F., Garín, M. A. and Unzueta, A. (2016). Cluster lagrangean decomposition in multistage stochastic optimization, *Computers & Operations Research* **67**: 48–62.

Fahle, T., Schamberger, S. and Sellmann, M. (2001). Symmetry breaking, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 93–107.

Farahani, R. Z., Hekmatfar, M., Fahimnia, B. and Kazemzadeh, N. (2014). Hierarchical facility location problem: Models, classifications, techniques, and applications, *Computers & Industrial Engineering* **68**: 104–117.

FICO Xpress (2016). `http://www.fico.com/en/products/fico-xpress-optimization`. Accessed: 2018-08-01.

Forrest, J., Ralphs, T., Vigerske, S., LouHafer, Kristjansson, B., jpfasano, EdwinStraver, Lubin, M., Santos, H. G., rlougee and Saltzman, M. (2018). coin-or/cbc: Version 2.9.9. **URL:** *https://doi.org/10.5281/zenodo.1317566*

Fourer, R., Gay, D. M. and Kernighan, B. (1993). *Ampl*, Vol. 117, Boyd & Fraser Danvers, MA.

Frisch, A. M., Grum, M., Jefferson, C., Hernández, B. M. and Miguel, I. (2007). The design of essence: A constraint language for specifying combinatorial problems., *IJCAI*, Vol. 7, pp. 80–87.

Gamrath, G., Fischer, T., Gally, T., Gleixner, A. M., Hendel, G., Koch, T., Maher, S. J., Miltenberger, M., Müller, B., Pfetsch, M. E. et al. (2016). The scip optimization suite 3.2.

GAMS - A User's Guide (2017). `https://www.gams.com/24.8/docs/userguides/GAMSUsersGuide.pdf`. Accessed: 2018-08-01.

Gassmann, H. I. (2005). The smps format for stochastic linear programs, *Applications of stochastic programming*, SIAM, pp. 9–19.

Gent, I. P., Jefferson, C. and Miguel, I. (2006). Minion: A fast scalable constraint solver, *ECAI*, Vol. 141, pp. 98–102.

Gent, I. P. and Walsh, T. (1999). Csplib: a benchmark library for constraints, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 480–481.

Getoor, L., Ottosson, G., Fromherz, M. and Carlson, B. (1997). Effective redundant constraints for online scheduling, *AAAI/IAAI*, Citeseer, pp. 302–307.

Geyer, C. J. and Thompson, E. A. (1992). Constrained monte carlo maximum likelihood for dependent data, *Journal of the Royal Statistical Society. Series B (Methodological)* pp. 657–699.

GLPK (2006).
URL: *http://www.gnu.org/software/glpk*

Gomes, C. P., Sabharwal, A. and Selman, B. (2007). Near-uniform sampling of combinatorial spaces using xor constraints, *Advances In Neural Information Processing Systems*, pp. 481–488.

Guesgen, H. W. and Hertzberg, J. (1992). *A perspective of constraint-based reasoning: an introductory tutorial*, Springer.

Gurobi Optimization Inc. (2014). Gurobi Optimizer Reference Manual. http://www.gurobi.com.
URL: *http://www.gurobi.com*

Hart, W. E., Laird, C. D., Watson, J.-P., Woodruff, D. L., Hackebeil, G. A., Nicholson, B. L. and Siirola, J. D. (2012). *Pyomo-optimization modeling in python*, Vol. 67, Springer.

Haugen, K. K., Løkketangen, A. and Woodruff, D. L. (2001). Progressive hedging as a meta-heuristic applied to stochastic lot-sizing, *European Journal of Operational Research* **132**(1): 116–122.

Hebrard, E. and Siala, M. (2017). Solver engine.

Heitsch, H. and Römisch, W. (2003). Scenario reduction algorithms in stochastic programming, *Computational optimization and applications* **24**(2-3): 187–206.

Hemmi, D., Tack, G. and Wallace, M. (2017). Scenario-based learning for stochastic combinatorial optimisation, *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer, pp. 277–292.

Hemmi, D., Tack, G. and Wallace, M. (2018). A recursive scenario decomposition algorithm for combinatorial multistage stochastic optimisation problems, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*.
URL: *https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16699*

Hochreiter, R. (2016). Modeling multi-stage decision optimization problems, *Computational Management Science*, Springer, pp. 209–214.

Hooker, J. N. and Ottosson, G. (2003). Logic-based benders decomposition, *Mathematical Programming* **96**(1): 33–60.

Høyland, K. and Wallace, S. W. (2001). Generating scenario trees for multistage decision problems, *Management science* **47**(2): 295–307.

IBM CPLEX (2011). *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*.

Infanger, G. (1999). Gams/decis user's guide.

Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming, *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, pp. 111–119.

Jorion, P. (2000). Value at risk.

Kallrath, J. (2013). *Modeling languages in mathematical optimization*, Vol. 88, Springer Science & Business Media.

Kim, S., Pasupathy, R. and Henderson, S. G. (2015). A guide to sample average approximation, *Handbook of simulation optimization*, Springer, pp. 207–243.

Kleywegt, A. J., Shapiro, A. and Homem-de Mello, T. (2002). The sample average approximation method for stochastic discrete optimization, *SIAM Journal on Optimization* **12**(2): 479–502.

Kotthoff, L. (2016). Algorithm selection for combinatorial search problems: A survey, *Data Mining and Constraint Programming*, Springer, pp. 149–190.

Laporte, G. and Louveaux, F. V. (1993). The integer l-shaped method for stochastic integer programs with complete recourse, *Operations Research Letters* **13**(3): 133 – 142. **URL:** *http://www.sciencedirect.com/science/article/pii/016763779390002X*

Lauriere, J.-L. (1978). A language and a program for stating and solving combinatorial problems, *Artificial intelligence* **10**(1): 29–127.

Lei, S., Wang, J., Chen, C. and Hou, Y. (2016). Mobile emergency generator pre-positioning and real-time allocation for resilient response to natural disasters, *IEEE Transactions on Smart Grid* .

Løkketangen, A. and Woodruff, D. L. (1996). Progressive hedging and tabu search applied to mixed integer (0, 1) multistage stochastic programming, *Journal of Heuristics* **2**(2): 111–128.

Maddala, G. S. and Lahiri, K. (1992). *Introduction to econometrics*, Vol. 2, Macmillan New York.

Maron, O. (1994). *Hoeffding Races–model selection for MRI classification*, PhD thesis, Massachusetts Institute of Technology.

MOSEK (2016). `http://docs.mosek.com/7.1/toolbox/index.html`. Accessed: 2018-08-01.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. (2007). Minizinc: Towards a standard cp modelling language, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 529–543.

Ohrimenko, O., Stuckey, P. J. and Codish, M. (2009). Propagation via lazy clause generation, *Constraints* **14**(3): 357–391.

Opturion CPX (2018). `https://www.opturion.com/about`. Accessed: 2018-08-01.

Powell, W. B. (2014). Clearing the jungle of stochastic optimization, *Bridging data and decisions*, Informs, pp. 109–137.

Prestwich, S. and Beck, J. C. (2004). Exploiting dominance in three symmetric problems, *Fourth international workshop on symmetry and constraint satisfaction problems*, pp. 63–70.

Prestwich, S. D., Tarim, S. A., Rossi, R. and Hnich, B. (2015). Hybrid metaheuristics for stochastic constraint programming, *Constraints* **20**(1): 57–76.

Prestwich, S., Rossi, R., Tarim, S. and Visentin, A. (2018). Towards a closer integration of dynamic programming and constraint programming.

Proll, L. and Smith, B. (1998). Integer linear programming and constraint programming approaches to a template design problem, *INFORMS Journal on Computing* **10**(3): 265–275.

Prud'homme, C., Fages, J.-G. and Lorca, X. (2014). Choco documentation, *TASC, INRIA Rennes, LINA CNRS UMR* **6241**.

Rahmaniani, R., Crainic, T. G., Gendreau, M. and Rei, W. (2017). The benders decomposition algorithm: A literature review, *European Journal of Operational Research* **259**(3): 801 – 817.
URL: *http://www.sciencedirect.com/science/article/pii/S0377221716310244*

Rendl, A., Guns, T., Stuckey, P. J. and Tack, G. (2015). Minisearch: a solver-independent meta-search language for minizinc, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 376–392.

Rendl, A., Tack, G. and Stuckey, P. J. (2014). Stochastic minizinc, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 636–645.

Robinson, S. M. (1996). Analysis of sample-path optimization, *Mathematics of Operations Research* **21**(3): 513–528.
URL: *https://doi.org/10.1287/moor.21.3.513*

Rockafellar, R. T. and Wets, R. J.-B. (1991). Scenarios and policy aggregation in optimization under uncertainty, *Mathematics of operations research* **16**(1): 119–147.

Rossi, F., Van Beek, P. and Walsh, T. (2006). *Handbook of constraint programming*, Elsevier.

Rossi, R., Hnich, B., Tarim, S. A. and Prestwich, S. (2015). Confidence-based reasoning in stochastic constraint programming, *Artificial Intelligence* **228**: 129–152.

Rubinstein, R. Y. and Shapiro, A. (1990). Optimization of static simulation models by the score function method, *Mathematics and Computers in Simulation* **32**(4): 373 – 392.
URL: *http://www.sciencedirect.com/science/article/pii/0378475490901426*

Ryan, K., Ahmed, S., Dey, S. S. and Rajan, D. (2016). Optimization driven scenario grouping.

Ryan, K., Rajan, D. and Ahmed, S. (2016). Scenario decomposition for 0-1 stochastic programs: Improvements and asynchronous implementation, *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, IEEE, pp. 722–729.

Sandikci, B. and Özaltın, O. Y. (2014). A scalable bounding method for multi-stage stochastic integer programs.

Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H. and Stuckey, P. J. (2013). Search combinators, *Constraints* **18**(2): 269–305.

Schulte, C., Lagerkvist, M. and Tack, G. (2006). Gecode, *Software download and online material at the website: http://www. gecode. org* pp. 11–13.

Schultz, R. (2003). Stochastic programming with integer variables, *Mathematical Programming* **97**(1-2): 285–309.

Shapiro, A. (2013). Sample average approximation, *Encyclopedia of Operations Research and Management Science*, Springer, pp. 1350–1355.

Shapiro, A., Dentcheva, D. and Ruszczyński, A. (2009). *Lectures on stochastic programming: modeling and theory*, SIAM.

Shapiro, A. and Philpott, A. (2007). A tutorial on stochastic programming, *Manuscript. Available at www2. isye. gatech. edu/ashapiro/publications. html* **17**.

Sheldon, D., Dilkina, B., Elmachtoub, A. N., Finseth, R., Sabharwal, A., Conrad, J., Gomes, C. P., Shmoys, D., Allen, W., Amundsen, O. et al. (2012). Maximizing the spread of cascades using network design, *arXiv preprint arXiv:1203.3514* .

Sutherland, I. E. (1964). Sketch pad a man-machine graphical communication system, *Proceedings of the SHARE design automation workshop*, ACM, pp. 6–329.

Tarim, S. A., Manandhar, S. and Walsh, T. (2006). Stochastic constraint programming: A scenario-based approach, *Constraints* **11**(1): 53–80.

Tarim, S. A. and Miguel, I. (2005). A hybrid benders' decomposition method for solving stochastic constraint programs with linear recourse, *International Workshop on Constraint Solving and Constraint Logic Programming*, Springer, pp. 133–148.

Thorsteinsson, E. (2001). Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming, *Principles and Practice of Constraint Programming—CP 2001*, Springer, pp. 16–30.

Valente, C., Mitra, G., Sadki, M. and Fourer, R. (2009). Extending algebraic modelling languages for stochastic programming, *INFORMS Journal on Computing* **21**(1): 107–122.

Van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language, *Journal of the ACM (JACM)* **23**(4): 733–742.

Van Hentenryck, P. (1999). *The OPL optimization programming language*, Mit Press.

Van Hentenryck, P., Michel, L., Perron, L. and Régin, J.-C. (1999). Constraint programming in opl, *International Conference on Principles and Practice of Declarative Programming*, Springer, pp. 98–116.

Van Slyke, R. M. and Wets, R. (1969). L-shaped linear programs with applications to optimal control and stochastic programming, *SIAM Journal on Applied Mathematics* **17**(4): 638–663.

Vázsonyi, M. (2006). Overview of scenario tree generation methods, applied in financial and economic decision making, *Periodica Polytechnica. Social and Management Sciences* **14**(1): 29.

Walsh, T. (2002a). Stochastic constraint programming, *ECAI*, Vol. 2, pp. 111–115.

Walsh, T. (2002b). Stochastic opl, *Proceedings of the Workshop on Modelling and Solving with Constraints*, Citeseer.

Walsh, T. (2006). General symmetry breaking constraints, *International Conference on Principles and Practice of Constraint Programming*, Springer, pp. 650–664.

Walsh, T. (2010). Symmetry within and between solutions, *Pacific Rim International Conference on Artificial Intelligence*, Springer, pp. 11–13.

Watson, J.-P. and Woodruff, D. L. (2011). Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems, *Computational Management Science* **8**(4): 355–370.

Watson, J.-P., Woodruff, D. L. and Hart, W. E. (2012). Pysp: modeling and solving stochastic programs in python, *Mathematical Programming Computation* **4**(2): 109–149.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization, *IEEE transactions on evolutionary computation* **1**(1): 67–82.

Wu, X., Xue, Y., Selman, B. and Gomes, C. P. (2017). Xor-sampling for network design with correlated stochastic events, *arXiv preprint arXiv:1705.08218* .

Vita

Publications arising from this thesis include:

David Hemmi, Guido Tack, and Mark Wallace

"Scenario-based learning for stochastic combinatorial optimisation". *Integration of AI and OR Techniques in Constraint Programming*, CPAIOR *2017, Proceedings, pp. 277-292.*

David Hemmi, Guido Tack, and Mark Wallace

"Decomposition-Based Solving Approaches for Stochastic Constraint Optimisation". *Thirty-Second Conference on Artificial Intelligence*, AAAI *2018, Proceedings, pp. 1322-1329.*

Permanent Address: Caulfield School of Information Technology
Monash University
Australia

This thesis was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.