Finding Diverse Solutions in Constraint Programming with Probabilistic Approaches

Mathieu Vavrille

COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

Nantes Université

THÈSE DE DOCTORAT DE

NANTES UNIVERSITÉ

ÉCOLE DOCTORALE N° 641
Mathématiques et Sciences et Technologies du numérique de l'Information et de la Communication
Spécialité : *Informatique*

Par
Mathieu Vavrille

Finding Diverse Solutions in Constraint Programming with Probabilistic Approaches

En vue de la soutenance de Thèse à Nantes, LS2N, site FST, bât 11, salle 3, le 14 septembre 2023
Unité de recherche : UMR 6004 - LS2N

Rapporteurs avant soutenance :

Özgür AKGÜN	Lecturer	School of Computer Science, Saint Andrews, United Kingdom
Claude-Guy QUIMPER	Professeur	Université Laval, Québec, Canada

Composition du Jury :

Président :	Eric MONFROY	Professeur des universités	Université Angers, Angers, France
Examinateurs :	Özgür AKGÜN	Lecturer	School of Computer Science, Saint Andrews, United Kingdom
	Emmanuel HEBRARD	Maître de conférence	LAAS, Toulouse, France
	Colin DE LA HIGUERA	Professeur des universités	Nantes Université, Nantes, France
	Claude-Guy QUIMPER	Professeur	Université Laval, Québec, Canada
Dir. de thèse :	Charlotte TRUCHET	Maîtresse de conférence HDR	Nantes Université, Nantes, France
Co-enc. de thèse :	Charles PRUD'HOMME	Maître de conférence	IMT Atlantique, Nantes, France

Acknowledgement

I first of all want to thank my supervisors, Charlotte and Charles for these amazing three years of research. You two make an exceptional team. With Charles I learned so much about implementation and CP solvers. I now love `choco-solver` and I am extremely proud of having contributed to it. Charlotte, you introduced me to CP and now I can't stop thinking in terms of constraints or optimisation problems (I love it).

I am honored to have such a jury for my defense. I want to thank the reviewers Claude-Guy and Özgür for their valuable comments on the manuscript that make me proud of my work. Emmanuel, your work inspired me throughout my thesis. I want to thank Eric and Marie for being members of my CSI, and Colin for the internship I did with you and the popularisation opportunity.

Obviously, I want to thank Marie for everything. You probably don't understand anything, but you still listen and your comments often help me to solve bugs. I am sorry to wake up so late everyday, or to stay up late to code or play. I want to thank all my family, I know that I don't call enough, but all the holidays with you are the times where I can truly rest. Anaïs and Julien (my favourite coop player), we are now far apart, but I hope that we will always keep in touch.

Finally, I have to thank all my fellow PhD students for all the cheering during the good times, and all the help during the bad times. So thank you to all the LOGIN association, Rémi, Anna, Marinna, Charles, Mathieu, Josselin, David, Thibault for all the fun during lunch; François, Louis, Léo for the distant gaming sessions; Giovanni, Pierre, Ghiles, Erwan for the scientific and professional help. If you are not in this list, you are probably in the following mosaic of Discord icons, so thank you for these three years.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Repetition is often boring. Reading a book for the second time is not the same, we already know the plot, the ending and everything that happens. In a theme park, the second time you ride a roller coaster is not the same. After each ride, the amount of emotion diminishes. Fortunately, there are often other roller coasters in the same park, and there are many new books to read.

When creating something (a new roller coaster, a book, music), the creators try to find new ways put together elements that have never been seen before. They try to make something new out of what already exists. In recent years, many Artificial Intelligence (AI) tools have been developed and improved to assist creation in many areas. For example, one of my favourite electronic music creators, DJ S3RL, created a song using only AI. He made a video about the whole process [1]. He trains OpenAI Jukebox [169] by giving it all his tracks. OpenAI Jukebox then generates a new music. This music should be edited manually by DJ S3RL, as it has good melodies but lacks the musical structure (fixed BPM, repetition of a chorus). Also, the lyrics generated are not intelligible. DJ S3RL then uses another AI to generate the lyrics, and uses *Vocaloid* to make an AI sing the lyrics. Some post-processing is required to add stereo tracks and bass, and to fine-tune the music. The music video is also created by an AI. Interestingly, S3RL is not a computer scientist. This means that all the tools he uses can be used by non-experts. All these AI tools give very diverse results, so they can be used by a content creator to get new ideas that can then be improved by hand.

1. `https://www.youtube.com/watch?v=JChbUcjZUBM`

1.1 Examples in Games

As a fan of board games and video games, I want to find new games with new ideas to play something I have never played before [2]. But good games are often designed to be replayable. This means that some elements of the game must change each time we play it.

A bad example of a game is the Tic Tac Toe. This game may be fun to play a few times, but after a while, a simple strategy can be found because there are only a few possible moves, there is no difference between the games. There is no reason to play this game again.

Interestingly, there are not that many ways to make something replayable.

1.1.1 Deterministic, but Combinatorial Games

Tic Tac Toe is a deterministic game with complete information. Its weakness comes from the fact that all games can be easily enumerated by hand (up to symmetry). However, other games are also deterministic with complete information, but are highly replayable. This is the case of chess or go. In chess, the first few moves are often memorised in order to avoid big mistakes. After the opening, however, the game leads to a board that has never been seen before in any game in history. This is where the replayability comes in. The game is now completely new, and it is not possible to rely on a pre-determined strategy.

Many video games are deterministic (i.e. giving the same inputs to the game will result in the same behaviour). An example of such a game is Trackmania [3], a highly competitive arcade racing game. This game runs at 60 frames per second, so the player's inputs are recorded every 1/60-th of a second. No matter how good the players are, no one can perform consistent actions every hundredth of a second (these are known as frame perfect tricks, and are extremely difficult to perform even once). In Trackmania, after a few seconds, the player will not have performed exactly the same movements, and will therefore be in a new position and have to react to other elements of the track.

Logic games are a special type of deterministic and combinatorial game. Also called puzzles, they are one-player games in which from an initial position there is a single solution satisfying the rules of the game. Often, the rules are defined as constraints on the

2. I focus on games with complete information. In games with incomplete information, also called *fog of war* in video games, uncertainty should be taken into account.

3. `www.trackmania.com/`

solution. For example, in *Sudoku* a solution contains different values in the 9 cells, rows and columns. There are several different games, often involving numbers. I developed an Android app [4] that solves some logic games (Sudoku, Kakuro, Slitherlink, Bridges, and Rikudo). In this app, I use constraint programming to find the solution to a logic game, as the rules of the game are constraints on the solution.

Replayability in logic games comes from the different starting positions. Sometimes, *patterns* can be used to solve the game, so diverse patterns should be present in the starting position such that different techniques are used to solve the game. Creating a starting position for a logic game is not an easy task. There should be only one solution from this starting position. Also, there are often several levels of difficulty, depending on the techniques used when solving by hand.

1.1.2 Procedurally Generated Worlds

Creating the world in which the players move is a difficult task. Procedural generation makes this task easier by allowing an algorithm to do the world generation. In a game, the world is designed by the developers, so there is a limit to the size of the world that can be created in a given amount of time. To generate diverse maps, the game designers can define some basic blocks and an algorithm will generate a world from these basic blocks. For example, the designers can create different houses, buildings, and wall textures (brick, concrete). Then an algorithm generates different city layouts, with buildings and houses with different textures. This can either be used as a first step for designers to improve later, or it can be the whole world, which can thus be much larger than hand-made worlds.

Examples of video games that use such a procedure are *Minecraft* and many *rogue-like* games [5]. In these games, the world is procedurally generated to be different each time. For example, in *The Binding of Isaac* each room is handmade, but the layout of the maze is procedurally generated. In *Dead Cells*, small layouts are designed, and glued together by an algorithm at each restart. In the board game *Magic Maze*, the maze is generated as it is discovered, by adding tiles to the board.

4. `https://play.google.com/store/apps/details?id=com.mvavrill.logicGamesSolver`
5. In *rogue-like* games, the game is restarted from scratch each time the player dies

1.1.3 Randomised Games

Procedurally generating a world provides an efficient way to create large maps. However, if the algorithm is deterministic, the generated world would always be the same. For multiple generation, randomness is used to change the output.

Randomness can also be used as a central mechanic. Dice games are an example of such use of randomness. In the game *Yahtzee* (or *Yams*), five dice are thrown to make combinations. However, using only randomness, there is not much to play with. For this reason, the dice can be rolled two more times to change the outcome. *Dice Forge* is also a dice game, but the dice used are special. The sides of the dice can be changed (by buying new sides). The new sides can contain more items (coins, or other game currency), so new dice rolls will produce more coins new rolls, for example, more coins. In card games, the deck is often shuffled at the beginning of the game, changing the order in which the cards can be played.

In some of the most popular video games, some players have developed a way to increase replayability by introducing randomness into the game. We focus on Metroidvania style games where unlocking power-ups gives access to new parts of the map. For example, unlocking the ability to jump allows the player to access new areas and find new objects. Examples of such games are the *The Legend of Zelda* series, *Metroid* and *Hollow Knight*. Some players have created *randomisers* for these games, a tool that randomises the position of the objects on the map. However, if the jump ability is behind a gap, or above the player it would not be possible to get it, and the player would be stuck. In Hollow Knight, for each location, the players made a list [6] of the items needed to reach that location. This gives constraints on the possible items in a given location. The problem then becomes finding a way to place the items in the locations, in a random way, while allowing the game to be completed, i.e. while satisfying the constraints. This problem of finding a solution of a combinatorial structure can be very hard to tackle without the right tools [172]. In Hollow Knight, the randomiser uses a heuristic approach [7]. This approach can fail, so restarts are made until a solution is found.

6. `https://github.com/homothetyhk/RandomizerMod/tree/master/RandomizerMod/Resources/Logic`
7. `https://github.com/homothetyhk/RandomizerMod/tree/master#progression-depth-settings`

1.2 Constraint Programming

Constraint Programming (CP) as an AI declarative programming technique. With a declarative programming technique, the properties on the solutions are defined, but the actual algorithm finding the solutions is already implemented. In constraint programming, the user declares *constraints* that should be satisfied by a set of *variables*, and a CP *solver* finds a solution (i.e. a value for the variables) that satisfies the constraints. Constraint programming is very generic, with several constraints allowing the user to state high level properties on the solutions. It can be seen as a black-box: the user inputs constraints, and the algorithms finds a solution, but the search process can also be tuned in multiple ways. In that sense, it can be referred to as a grey-box, i.e. the main algorithm cannot be modified, but an extensive API is provided to tweak the behaviour.

We distinguish between three types of people working with CP solvers. At one end, the users have real-life problems to solve. In Constraint Programming, we think in terms of solutions: what do the users want to get out of the algorithm and how can a *solution* can be described. Once the users have described their problem (this can be an iterative process), a modeller translates this problem into a constraint satisfaction problem, i.e. in the CP solver language. This modeller should have knowledge of the functions (and constraints) provided by the solver (either through an API, or using high level languages such as MiniZinc [39] or XCSP^3 [11]). In this step, modelling choices can be made, and search strategies can be defined to implement domain knowledge in the CP solver. At the other end of the CP application, the CP solver developer implements the tools required by the modeller to find the solutions. The developer should provide an easy to use API, but also implement all the efficient constraint propagation algorithms in the back-end of the solver.

In this thesis, we oscillate between the modeller and the developer, while still considering the needs of the end-users. We want to define diversity in a way that is easy to use for a modeller, and implement it in the solver. For example, in Chapter 5, we provide a way to generate solutions randomly, and implemented it in the solver.

1.3 Contributions

This thesis is about solution diversity in CP solvers by using probabilistic approaches. The backtrack search of CP solvers is a powerful, but rigid framework for finding solutions.

We propose ways to tweak the behaviour of the solver using randomness to generate diverse solutions. We also thoroughly analyse the behaviour of our algorithms (and state-of-the-art algorithms) to understand their properties.

A review of state-of-the-art constrained samplers We review constrained state-of-the-art samplers and evaluation tools. We present the samplers (with the pseudo-code, or an outline of it) in such a way that the differences between them and the improvements over the years can easily be understood. This allow to have a clear overview of the field of sampling in constrained problems (in SAT and CP).

A new sampler for CP problems We propose a new sampler, TableSampling, dedicated to constraint programming problems. It is the first CP sampler that works in the CP framework. TableSampling is now available in the solver `choco-solver` (since version 4.10.9).

A strategy for diversification in pattern mining We present a new search strategy dedicated to pattern mining, OrientedSearch, which is used to orient the search towards spaces with diverse solutions. We also show that the default random search strategy is a very fast approach that returns diverse solutions.

A strategy for t-wise coverage in feature models We propose a second search strategy, FrequencyDiff, dedicated to the generation of high t-wise coverage test suites. This search strategy greatly improves the size and the quality of the generated test suites.

A probabilistic study of the RandomSearch search strategy We analyse the behaviour of the default search strategy RandomSearch on the t-wise coverage problem. We show a lower bound on the probability of drawing the t-wise combinations when using RandomSearch.

Diversity constraints and algorithms We prove multiple properties on the diversity constraints, depending on the aggregator and distance used. We also prove properties on the greedy and random algorithms.

An application to a multi-objective real life problem We apply diversity to a multi-objective application. We show a two-step approach adapted to the multi-objective framework to generate good diverse solutions.

1.4 Outline

First, in Part I we define the background of the thesis. Chapter 2 formally defines Constraint Programming. We present it in two steps: first from a user/modeller point of view, i.e. as a declarative programming framework, and then we present the general solving algorithm. We also present the SAT framework (mostly used in Chapters 4 and 7) and the diversity definitions in CP (mostly used in Chapter 6 and Part IV). The following Chapter 3 defines the probability concepts used in this thesis. First, we recall the classical definitions and notations. These definitions and notations are used in every chapter of this thesis. We also introduce the hashing framework, by defining *hashing constraints*. These hashing constraints are used by several samplers in Part II.

The following chapters of this thesis are contributions. The second part, Part II, focuses on samplers. First, in Chapter 4 we present several state-of-the-art constrained samplers. Then, in Chapter 5, we present a new sampler that we have designed: TableSampling. This chapter is mostly taken from our conference [1] and journal [2] publications.

The third part, Part III, presents two uses of constraint programming to generate diverse solutions using search strategies. Chapter 6 presents our contribution on pattern mining. Then, in Chapter 7, we present our contribution on feature models, both theoretical and practical. A part of this work on feature models (Section 7.6) has been published as a research report [3].

In the fourth and final part of this thesis, Part IV, we study diversity approaches in CP. In Chapter 8 we study in detail the diversity constraints and the properties and guarantees of the algorithms. In Chapter 9 we study diversity in a multi-objective application.

Appendices are presented at the end of this thesis in Part V. First, Appendix A contains some technical proofs of Chapters 7 and 8. Appendix B presents a supplementary material for Chapter 5. Appendix C presents the Android application I developed, *Logic Games Solver*, with CP models for several logic games.

Part I

Background

Chapter 2

Constraint Programming

2.1 Introduction

Constraint Programming (CP) follows the declarative programming paradigm. As such, it can be used as a black-box. However, CP solvers also propose several ways to tune the search, either to speed-up the process or to find different solutions.

In this chapter, we first introduce constraint satisfaction problems from a modelling point of view in Section 2.2. As an example, we present the model of a logic game, rikudo, in Section 2.3. We present how solutions are found in solvers in Section 2.4, i.e. from the solver's point of view. An overview of SAT solving is given in Section 2.5. In Section 2.6 we define the diversity in the case of combinatorial problems.

2.2 Definitions

As a declarative programming paradigm, Constraint Programming focuses on solutions. In this section, we consider CP from a user's point of view, i.e. from a modelling point of view. We define what are solutions and how to define constraints on those solutions. Constraint Satisfaction Problems (CSP) provide the framework for defining problems.

Definition 1 (Constraint Satisfaction Problem (CSP))**.** A CSP $\mathcal{P}$ is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables. These variables are the unknowns of the problem;
- $\mathcal{D}$ is a function that associates a domain with each variable. In this thesis, the domains are a finite subset of the integers;
- $\mathcal{C}$ is a set of constraints, each constraint $C \in \mathcal{C}$ consisting of:

— a tuple of variables called *scope* of the constraint $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, where r is the arity of the constraint. It defines the variables involved in the constraint.
— a relation, i.e. a set of instantiations

$$rel(C) \subseteq \prod_{k=1}^{r} \mathcal{D}(X_{i_k}) \ .$$

The relation defines the values that the variables of the scope can take.

If a variable has a domain $\mathcal{D}(X) = \{0, 1\}$, it is said to be Boolean. When all the variables of the problem are Boolean and the constraints are propositional formulae, the framework of SAT can be used, presented in section 2.5. The domains of the variables define a *search space*, i.e. all the values that can be taken by all the variables: $\prod_{i=1}^{n} \mathcal{D}(X_i)$.

2.2.1 Solutions

Within the search space, the *solution space*, which contains all the solutions to their problem, is the one that users are interested in.

Definition 2 (Solution)**.** Let $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. We call *instantiation* a function $\sigma : \mathcal{X} \to \bigcup_{X \in \mathcal{X}} \mathcal{D}(X)$ which associates each variable X to a value in its domain $\mathcal{D}(X)$, i.e. $\forall X \in \mathcal{X}, \sigma(X) \in \mathcal{D}(X)$.

An instantiation σ is said to *satisfy* a constraint C if the values associated with each variable of the scope $scp(C)$ are in the relation $rel(C)$, i.e. if $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, then $(\sigma(X_{i_1}), \ldots, \sigma(X_{i_r})) \in rel(C)$.

An instantiation is said to be a *solution* if it satisfies *all* the constraints in $\mathcal{C}$. We note $Sols(\mathcal{P})$ the set of solutions to the problem $\mathcal{P}$, also called the *solution space*.

Once a solution is found, it can be presented to a user, for example as a schedule (in the case of a scheduling problem), or as routes on a map (in the case of a vehicle routing problem), etc. Depending on the user, *solving* a problem can have different meanings. Users may want a single solution, or all solutions. They may also want some solutions that they can compare and choose from. On the other hand, solutions may be associated to an objective function, assessing how acceptable the solution is. In this case, solving the problem is finding the solution that optimises the given criterion.

Definition 3 (Constraint Optimisation Problem (COP))**.** A COP is a quadruplet $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, obj \rangle$ where obj is a special variable, called the *objective* to be optimised. Then,

solving the COP means finding a solution σ_{opt} that maximises the variable obj, i.e.

$$\sigma_{opt} = \underset{\sigma \in Sols(\mathcal{P})}{\operatorname{argmax}} \sigma(obj)\,.$$

Remark. *We have presented the definition as a maximisation problem, but minimisation is also allowed in COPs.*

When there are multiple (possibly conflicting) objectives, the problem is called a *multi-objective* problem. In this case it is more difficult to define what is the "best" solution. Multi-objective problems and solution quality are defined and studied in detail in Chapter 9.

2.2.2 Constraints

In the definition 1 of CSPs, constraints are defined by a scope and a relation. This definition of a constraint in extension (by listing the allowed values) is called a `table` constraint.

Definition 4 (Table constraint)**.** Given a tuple of r variables $X_{i_1}, \ldots, X_{i_r}$, and a set of tuples $\mathcal{T}$, the table constraint $C = \texttt{table}((X_{i_1}, \ldots, X_{i_r}), \mathcal{T})$ is such that $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, and $rel(C) = \mathcal{T}$.

Table constraints allow the representation of any constraint or relationship between variables. However, it is not user-friendly, as the users have to determine themselves all the allowed values, and in the worst case, the number of tuples in rel can be exponential in the number of variables in the table. CP languages (such as MiniZinc [39] or XCSP^3 [11]) allow for a wide range of constraints defined in intension. These constraints ease the modelling phase, but also helps the solver, as they are often associated to dedicated algorithms. For example, the arithmetic constraint $X + Y \leq 2$ (with $X, Y \in \{0, 1, 2\}$) is a condensed representation of the constraint C such that $scp(C) = \{X, Y\}$ and $rel(C) = \{(0,0), (0,1), (0,2), (1,1), (1,2), (2,0)\}$. This language also includes *global constraints*, which are predicates that express a conjunction of several other constraints.

Definition 5 (Global constraint (from Chapter 6 of [50]))**.** A *global constraint* is a constraint that captures a relationship between a non-fixed number of variables.

Global constraints facilitate the modelling phase. There are many global constraints to model different behaviours (423 global constraints in the global constraint catalogue [1] at the time of writing). One of the most classic constraint is the `alldifferent`.

Definition 6. The `alldifferent` constraint ensures that all the variables take different values, i.e. if σ is an instantiation then

$$\sigma \text{ satisfies } \texttt{alldifferent}(X_{i_1}, \ldots, X_{i_r}) \Leftrightarrow \forall 1 \leq j < k \leq r, \sigma(X_{i_j}) \neq \sigma(X_{i_k}) \,.$$

As we will see in the following section 2.4.1, global constraints not only make modelling easier, they also allow for better and faster algorithms.

2.3 Example of Model: Rikudo

App logo

In this section we present an example of problem, and the corresponding CSP. This example comes from the Android application I developed for solving logic games using CP. The detailed presentation of the application (called Logic Games Solver [2] and available in the Play Store) is given in Appendix C. There are several ways of representing a single problem. We present here one way to model this problem, but it is a good exercise to try to model it in a different way.

2.3.1 Game Rules

Rikudo is a logic game played on a hexagonal grid. Logic games are games where a set of rules and initial clues restrict the problem to a unique solution. In Rikudo, the goal is to enter numbers from 1 to M (where M is the number of cells, 36 in the example of Figure 2.1). These numbers must form a continuous path from 1 to M using adjacent cells. Some numbers are already given as clues. Also, some edges are required to be taken in the path.

Figure 2.1 shows an input (which you can solve). The small squares between two cells represent mandatory edges. Note that the middle cell is not used in this game.

1. `http://sofdem.github.io/gccat/`
2. `https://play.google.com/store/apps/details?id=com.mvavrill.logicGamesSolver`

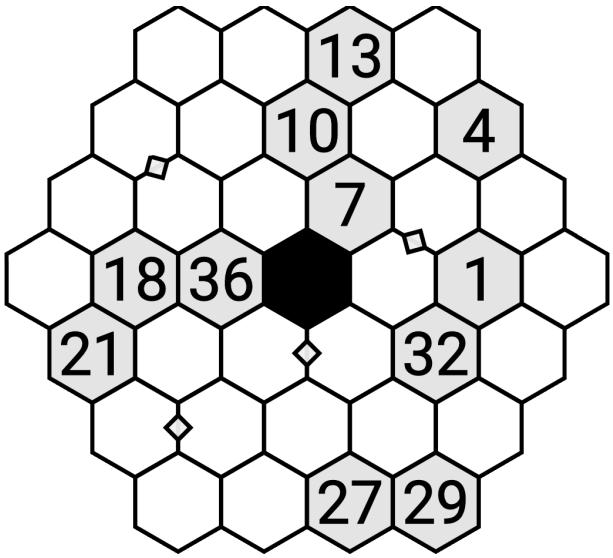


Figure 2.1 – Example of rikudo grid

2.3.2 Model

Here we define a CSP that can be used to solve this problem. Here we use one integer variable per cell, and Boolean variables representing the underlying graph from which a path is created.

Notation. *We note:*

- M *the number of cells. In the example* $M = 36$*.*
- S *the set of all hexagonal cells* $S = \{s_1, \ldots, s_M\}$ *(not in any order);*
- *Given a cell* $s \in S$*,* $N(s)$ *is the set of the neighbours of* s*, i.e. the 6 cells surrounding* s *(or less if* c *is on the border of the grid).*

We create M integer variables X_s, one for each cell:

$$\forall s \in S, X_s \in \{1, \ldots, M\}\,.$$

These are the variables that define the solution. A first constraint on these variables is that they should all be different. The constraint is

$$\texttt{alldifferent}(X_{s_1}, \ldots, X_{s_M})\,.$$

Remark. *If the values* 1 *and* 36 *are present as clues, this constraint is redundant with others presented later. A redundant constraint is not necessarily a bad thing, as it helps the solver.*

If a set of clues $\mathcal{H}^{cell}$ (for the values) is given containing pairs $(cell, value)$, we have to add the constraints

$$\forall (s, v) \in \mathcal{H}^{cell}, X_s = v\,.$$

Now we need to ensure that the clues form a single path, and that the values on that path are adjacent. We create Boolean variables representing directed edges from one cell to one of its neighbours:

$$\forall s \in S, \forall s' \in N(s), E_{s,s'} \in \{0, 1\}\,.$$

We use these variables to ensure that a path is taken through all the cells. We create a special cell s_0 which is connected to all other (real) cells, with the Boolean variables $E_{s_0,s}, E_{s,s_0}$ for $s \in S$. This special cell transforms the problem of finding a path into the problem of finding a cycle (by connecting the beginning and end of the path using s). Working with a cycle ease the modelling because there are no more special cases in the beginning and the end of the path. A cycle is defined by the fact that all cells have an out-degree and an in-degree equal to 1. This is enforced by the following constraints:

$$\begin{aligned}
\forall s \in S, \forall s' \in N(s) \cup \{s_0\}, \sum_{s' \in N(s) \cup \{s\}} E_{s,s'} &= 1 && \text{(out-edges)}\\
\forall s \in S, \forall s' \in N(s) \cup \{s_0\}, \sum_{s' \in N(s) \cup \{s\}} E_{s',s} &= 1 && \text{(in-edges)}\\
\sum_{s \in S} E_{s_0,s} &= 1 && \text{(special out-edges)}\\
\sum_{s \in S} E_{s,s_0} &= 1\,. && \text{(special in-edges)}
\end{aligned}$$

Remark. *If 1 is present as a clue, it is possible not to use the out-edges of the special cell, and to state that the sum of the in-edges going to the cell containing 1 is equal to 0. The same is true for the cell containing the clue M.*

We also need to use the clues on the mandatory edges. We assume that we have access

to a set $\mathcal{H}^{edge}$ containing pairs (s, s') of (undirected) mandatory edges. Then, the following constraint adds the clue constraints (one of the two directed edges has to be taken):

$$\forall (s, s') \in \mathcal{H}^{edge}, E_{s,s'} + E_{s',s} = 1$$

These constraints ensure that the underlying graph contains only cycles. However there can be more than one cycle. The following constraints, which link the edges to the values, deal with this issue, and at the same time constrain the values of the cells in the path. The constraint states that if the edge s, s' is taken, then the value of s' follows the value of s.

$$\forall s \in S, \forall s' \in N(s), E_{s,s'} \Rightarrow (X_{s'} = X_s + 1)\,.$$

The problem is now fully constrained, and if all the clues are given, only one solution is allowed.

Remark.

- *It is possible (to help the solver) to provide more expert knowledge by adding redundant constraints. For example, two non-adjacent cells cannot have adjacent values. Such a constraint can be defined as:*

$$\forall s \in S, \forall s' \notin N(s), X_{s'} \neq X_s + 1\,.$$

- *Other CSPs are possible to solve this problem. For example*[3]*, it is possible to use hexagonal coordinates, and define the path using moves in that coordinate system. Dealing with cell clues is very easy, but dealing with edge clues is harder.*

For more examples of models, the Appendix C presents CSPs for other logic games (such as Sudoku and Kakuro). I also show some optimisations that can be made by merging constraints, adding redundant constraints to help the solver, or using graph variables.

2.4 CP solving

In the previous section, I presented CP from the user's point of view. Here I present the solving process used to actually find the solutions. For actual practical implementations,

3. This idea comes from Matthew Coyle, a fellow PhD student who motivated me to solve this very interesting problem.

```
1 Function SOLVE(P)
      Data: A CSP P = ⟨X, D, C⟩
      Result: The set of solutions of P
2     P' ← PROPAGATE(P)
3     if ∃X_i ∈ X' such that |D'(X_i)| = 0 then
4         return ∅
5     else if ∀X_i ∈ X' with |D'(X_i)| = 1 then
6         return {D}
7     else
8         d ← MAKEDECISION(P')
9         return SOLVE(P' ∧ d) ∪ SOLVE(P' ∧ ¬d)
```

Algorithm 2.1: Recursive CP solver

every solver has a different framework, but an interested reader can look into [30], the inspiration for `choco-solver`.

Algorithm 2.1 presents a basic functional programming and recursive algorithm for finding all the solutions of a CSP. This algorithm alternates between two steps: propagation and decision. The propagation phase (line 2) analyses the constraints and the current domains, and tries to find values that cannot appear in solutions. These values can be safely filtered (i.e. removed from the domains). We present this step in detail in the following section 2.4.1. After this propagation step, if the domain of a variable is empty (line 3), then there are no solutions, the sub-problem is said to be *inconsistent*. If all the variables are *instantiated* (i.e. their domain contains a single value, in line 5), then a solution has been found and can be returned. Otherwise, if the sub-problem is consistent, and there are still uninstantiated variables, a decision must be performed. A function MakeDecision is called to generate a decision (a constraint that can be negated) in line 8. Then recursive calls are made on the sub-problem with the decision (and with its negation) in line 9. This decision step is described in detail in the following section 2.4.2.

The recursive calls in Algorithm 2.1 define a recursive tree. In CP, this tree is called the *backtrack tree*, and the *backtrack-search* traverses this tree. A backtrack is the action of returning from a sub-call of the Solve function. A backtrack occurs when a solution or an inconsistency is found, or when both branches of a decision have been traversed.

Remark. *This thesis is all about understanding the backtrack-search and trying to change the order in which solutions are returned. The backtrack-search has to completely enumerate a sub-space ($\mathcal{P}' \wedge d$) before moving on to the next sub-space ($\mathcal{P}' \wedge \neg d$). Hence, solutions close to each other are returned sequentially. If the search is stopped before all solutions*

have been enumerated, some search-spaces would not have been seen by the solver. Diversity properly defines how to evaluate whether a subset of solutions covers the space well, and is presented in Section 2.6.

2.4.1 Constraint Propagation

The first main component of the SOLVE function is the PROPAGATE function. Constraint propagation is the task of reducing the domain of the variables, without removing solutions. This reduces the search space, hence focusing the search on the solution space. This step searches for values that can be safely removed.

Consistency

In order to know which values can be deleted, a notion of consistency is defined. We present arc consistency, the most commonly used notion of consistency.

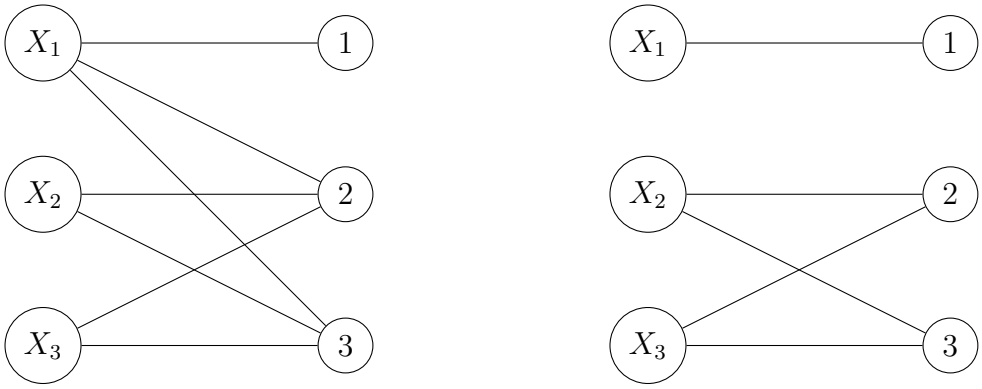
Definition 7 (Arc consistency [34])**.** Let $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. Let C be a constraint with $scp(C) = \{X_{i_1}, \ldots, X_{i_r}\}$. Let $j \in \{1, \ldots, r\}$, a value $x_{i_j} \in \mathcal{D}\left(X_{i_j}\right)$ is said to be *arc consistent* with constraint C iff there exists a tuple $\tau = (x_{i_1}, \ldots, x_{i_r})$ (with $x_{i_k} \in \mathcal{D}(X_{i_k})$) such that $\tau \in rel(C)$. τ is called a support for the value x_{i_j}.

If all the values in the domains of all variables are arc consistent with all constraints, then the CSP is also said to be arc consistent.

Intuitively, when considering one particular value and one constraint, if there is a value for all the other variables such that the constraint is satisfied, then the value under consideration is arc consistent. We illustrate arc consistency using the `alldifferent` constraint and its decomposition.

Example. *We consider the following CSP:*

$$\mathcal{P} = \begin{array}{l} \langle \quad \{X_1, X_2, X_3\}, \\ \qquad \mathcal{D} : \begin{cases} X_1 & \to \{1, 2, 3\} \\ X_2 & \to \{2, 3\} \\ X_3 & \to \{2, 3\} \end{cases}, \\ \qquad \{\mathit{alldifferent}(X_1, X_2, X_3)\} \ \rangle \end{array}$$



(a) Value graph (b) Value graph after filtering

Figure 2.2a shows a representation of the domains of the variables through the value graph*. This bipartite graph has nodes representing the variables, and nodes representing the values. An edge is present if the value is in the domain of the variable.*

In the CSP $\mathcal{P}$, we can check if the value 3 of X_2 is consistent (with the only constraint). The tuple $(1, 3, 2)$ satisfies the constraint, so the value 3 of X_2 is consistent. However, it is not possible to create a tuple satisfying the constraint with the value 2 (or 3) for X_1. If we try to instantiate X_1 to 2, then X_2 is necessarily instantiated to 3, and then there is no possibility for X_3. In consequence, the values 2 (and 3) can be removed from the domain of X_1, resulting in the value graph in Figure 2.2b.

Now we want to show that using the global constraint `alldifferent` *improves the propagation. This constraint can be decomposed into three $\neq$ constraints:$X_1 \neq X_2$, $X_1 \neq X_3$, and $X_2 \neq X_3$. We can check whether the value 2 for X_1 is consistent with this reformulation. It is consistent with the constraint $X_1 \neq X_2$ because the tuple $(2, 3)$ satisfies the constraint. The same reasoning works for the constraint with X_3, and X_1 does not appear in the last constraint.*

We see that the use of the global constraint allow us to find earlier that some values are inconsistent. We show another example of such constraint reformulation in the Kakuro logic game in Appendix C.4 where merging `alldifferent` *and* `sum` *constraints improves the propagation.*

Remark. *Other notions of consistency can be defined. For example, path consistency [6] is a stronger consistency, in the sense that it removes more values than arc consistency. On the other hand, bound consistency [13, 31, 47] filters fewer values (they restrict only the bounds of the domains). These definitions (as well as others) allow a stronger propagation at the cost of a longer running time, or fewer propagation but faster.*

```
1  Function PROPAGATE_AC3(P)
       Data: A CSP P = ⟨X, D, C⟩
       Result: false if the problem is inconsistent, true otherwise, with the
               domains reduced.
2      Q ← {(X, C) | C ∈ C, X ∈ scp(C)}
3      while Q ≠ ∅ do
4          (X, C) ← Q.pop()
5          if REVISE(X, C) then
6              if D(X) = ∅ then return false
7              else Q ← Q ∪ {(X', C') | C' ∈ C\{C}, X', X ∈ scp(C'), X' ≠ X}
8      return true
9  Function REVISE(P, X, C)
       Data: A CSP P = ⟨X, D, C⟩
       Result: false if the problem is inconsistent, true otherwise, with the
               domains reduced.
10     change ← false
11     for v ∈ D(X) do
12         if v is not arc-consistent with C then
13             Remove v from D(X)
14             change ← true
15     return change
```

Algorithm 2.2: AC3: enforcing arc-consistency.

Every constraint has its own filtering algorithm. Global constraints often allow to have very efficient algorithms to find values to filter. A textbook filtering algorithm is the one of the `alldifferent` constraint, presented in [49]. The author notes the correspondence between a support for a value, and a maximum matching in the value graph. He can then use matching theory to find which values that do not belong to any maximum matching.

Propagation Algorithm

When removing a value from a domain (using arc consistency), some other values may no longer be arc consistent with some other constraints. These values need to be checked on these constraints, until the problem becomes arc consistent. One of the first algorithms proposed to do this is AC3 [34], presented in Algorithm 2.2. It is based on a REVISE function that filters the values of X that are not arc consistent with C. It also informs the main algorithm when a change has been performed. The main algorithm

uses a queue Q of pairs of a variable X and a constraint C such that some values of X may not be arc consistent with the constraint C. While the queue is not empty, it pops a pair variable/constraint from the queue, and performs the consistency check using Revise. If a modification has been done, the algorithm checks that the domain is not empty (otherwise the sub-problem is inconsistent, line 6). It also updates the queue to add all the variables whose values may no longer be consistent. For all the constraints C such that $X \in scp(C)$, it adds the pair X', C for all $X' \in scp(C)$ (except X).

This algorithm was later improved, for example by storing more information, in AC4 [37], AC6 [7], and AC2001 [8]. These algorithms are very efficient when reasoning on the support tuples, i.e. with table constraints. Other frameworks for propagation have been proposed that focus on constraints [30, 55].

Implementing a CP solver is a difficult task because there are plenty of optimisations to consider. During the propagation phase, some constraints can be prioritised because they may filter more values faster (for example, the `alldifferent` may not filter many of values until the domains are small enough [10]). Algorithm 2.1 presented the CP solver in a recursive and functional programming pseudocode. Implementations may not use this framework. In this case, changes to domains should be recorded so that they can be undone later when a backtrack occurs.

2.4.2 Search Strategies

The second main component of the Solve function is the MakeDecision function. When no more constraint propagation can be done, a decision must be made to reduce the search space.

Definition

In all generality a decision is a constraint that can be negated. Such a constraint must be designed to directly reduce the domains of some variables. Otherwise, the search may get stuck in a loop of making decisions that do not reduce the domains. In most cases, the decisions are unary, i.e. their scope is a single variable. This way, the domain of the variable can be reduced immediately.

A unary constraint is necessarily of the form $X \in D$ with $D \subseteq \mathcal{D}(X)$ (and its negation $X \notin D$). Two particular sets D are interesting for decisions. If D contains all the values of $\mathcal{D}(X)$ smaller than some value v, the decision is equivalent to $X < v$ (and its negation

$X \geq v$). This decision is especially useful when dealing with continuous domains. If $|D| = 1$, the decision is directly an instantiation $X = v$, and its negation ($X \neq v$). In this manuscript, we will focus on this type of decision: $X = v$.

Remark. *In the* Solve *function, the branching is binary, i.e. the decision and its negation are applied. It is also possible to enumerate the values of a variable, and to make one recursive call per value.*

The search strategy, which chooses which decisions are made during the search, has a strong influence on the efficiency of the solver. Search strategies can be designed to perform well on a wide range of problems (black-box search strategies), or to perform well on a specific set of instances (such as scheduling, or routing problems). Some black box strategies are based on the domains of the variables. The `dom` [21] search strategy chooses the variable with the smallest domain. It can be improved with `dom/wdeg` [12], by weighting the constraints according to the conflicts they caused. To this day, `dom/wdeg` remains one of the most competitive black-box search strategies. Adaptive strategies, such as `impact` [48] and `activity` [36], collect and use information during the search, such as variables/values that lead to conflicts, or reduction of domains. `CBS` [42] uses counting algorithms on constraints to estimate the density of solutions, to guide the search towards promising spaces. There are meta-strategies that modify the behaviour of other strategies (for example by reducing the set of variables to branch on), such as `lastConflict` [32]. Search strategies are an active area of research, and newer strategies are often developed, such as `FRBA` [33] or $\texttt{wdeg}^{\texttt{ca.cd}}$ [63]. For some specific problems, a tailored strategy may improve the running time, such as `SetTimes` [18] for scheduling problems.

These search strategies are heuristics, and as such they may make decisions that do not lead to solutions. In this case, the solver may spend a long time in a sub-space without solution, before finally proving that there are no solutions. This phenomenon is called a heavy-tail [19] and can be avoided by using restarts. Restarts stop the search, and start again from the root of the search. Restarts can be performed after some conflicts (i.e. inconsistencies) have been detected. To avoid returning to a sub-space already traversed, no-goods can be added to the model. For example, the Luby [178] sequence defines how often restarts should be performed. Adaptive search strategies benefit from the use of restarts, because they learn from their mistakes. At the beginning of the search, bad decisions may be made, that are undone with restarts. After a while, the search learns from the conflicts, and makes better decisions (i.e. decisions that are likely to lead to solutions).

```
1 Function RandomSearch(𝒳)
      Data: 𝒳 the variables of the problem
      Result: A decision to perform in the search
2     𝒳' ← {X ∈ 𝒳 | |𝒟(X)| ≠ 1};
3     X ← Random(𝒳');
4     v ← Random(𝒟(X));
5     return Decision(X = v);
```

Algorithm 2.3: Random search strategy

RandomSearch

The search strategies presented so far are designed to find solutions quickly. They are often deterministic, i.e. running the solver twice will produce the same solution. In some cases it is interesting to run the solver a second time to get a different solution. In this case, randomness is useful.

Algorithm 2.3 presents RandomSearch, the basic random search strategy. This strategy considers all the variables, randomly chooses an uninstantiated variable X to branch on, randomly chooses a value v in its domain, and returns the decision $X = v$.

Using RandomSearch as the search strategy allows solutions to be returned randomly. However, some solutions are more likely to be selected than others.

Example. *We consider the following CSP:*

$$\mathcal{P} = \left\langle \{X, Y\}, \begin{cases} X & \mapsto \{0,1\} \\ Y & \mapsto \{0,1\} \end{cases}, \{X + Y > 0\} \right\rangle$$

This problem has three solutions for (X, Y)*:* $(0,1)$*,* $(1,0)$*, and* $(1,1)$*. We can now follow the behaviour of the solver to determine the probability of getting each solution. The initial propagation step cannot remove any value from the domain of the variables. The decision step is applied. A variable has to be chosen randomly, there are two cases (each with equal probability* $1/2$ *of being chosen):*

- *X is chosen, then one of its values is chosen at random, again there are 2 cases (each of equal probability* $1/2$ *of being chosen):*
  - *0 is chosen. The only possible solution resulting from this decision is the solution* $(X, Y) = (0,1)$ *(the propagation step will easily filter* 0 *out of the domain of* Y*);*
  - *1 is chosen. Then the constraint is satisfied, so no more filtering can be done.*

A decision step is performed again, selecting the only uninstantiated variable Y. Then there are two possible values:
— *0, then the solution is $(X, Y) = (1, 0)$;*
— *1, then the solution is $(X, Y) = (1, 1)$;*

- *Y is chosen, then one of its values is chosen at random, there are again 2 cases (each with equal probability $1/2$ of being chosen):*
  — *0 is chosen. The only possible solution resulting from this decision is the solution $(X, Y) = (1, 0)$ (the propagation step will easily filter 0 out of the domain of X);*
  — *1 is chosen. Then the constraint is satisfied, so no more filtering can be done. A decision step is performed again, choosing the only uninstantiated variable X. Then there are two possible values:*
  — *0, then the solution is $(X, Y) = (0, 1)$;*
  — *1, then the solution is $(X, Y) = (1, 1)$.*

Let us define s to be the random solution returned by the algorithm. The probability to return each solution can be computed. For example for $s = (0, 1)$, we have

$$\begin{aligned}\mathbb{P}(s = (0,1)) = &\quad \mathbb{P}(X \text{ is chosen first}) \cdot \mathbb{P}(0 \text{ is chosen for } X) \\ &+ \mathbb{P}(Y \text{ is chosen first}) \cdot \mathbb{P}(1 \text{ is chosen for } Y) \cdot \mathbb{P}(0 \text{ is chosen for } X) \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \\ &= \frac{3}{8}\end{aligned}$$

The final random distribution of the solutions when using the RandomSearch *search strategy is*

$$\begin{aligned}\mathbb{P}(s = (0,1)) &= \frac{3}{8} \\ \mathbb{P}(s = (1,0)) &= \frac{3}{8} \\ \mathbb{P}(s = (1,1)) &= \frac{1}{4}\end{aligned}$$

This distribution is not uniform, the solutions do not all have an equal chance of being sampled.

From this simple example we can see that the distribution of RandomSearch is

not uniform. This fact has also been proven experimentally in Chapter 5 Section 5.6. Computing the exact distribution of the solutions is not an easy task. In Chapter 7 Section 7.4, I analyse the behaviour of RandomSearch and give a lower bound on the probability of drawing certain combinations of variables.

Considering running time, RandomSearch is extremely simple to implement and fast to run (the only difficulty is knowing what are the uninstantiated variables). However, it makes very bad decisions. These bad decisions can lead the search into unsatisfiable sub-spaces for which the solver will take a long time to prove the unsatisfiability. This was shown experimentally on a benchmark of hard instances in Chapter 5 Section 5.7. On easier instances (either with many solutions or where the propagation quickly finds inconsistencies), it finds solutions much faster, as shown in the experimental section of both Chapters 6 and 7.

2.4.3 choco-solver

In this thesis, I used `choco-solver` [46] to implement the algorithms, and to solve the problems. `choco-solver` is an open-source CP solver implemented in `Java`, and available as a `Maven` dependency. There is also a `Python` binding [4]. The following code shows how to define variables and a simple constraint.

```
Model model = new Model("Example");
IntVar x = model.intVar("X",0,2);     // X \in {0,1,2}
IntVar y = model.intVar("Y",0,2);     // Y \in {0,1,2}
x.add(y).le(2).post();                // Constraint X+Y <= 2
Solver solver = model.getSolver();
solver.setSearch(Search.inputOrderLBSearch(new IntVar[]{x,y}));
solver.limitTime(1000);               // Time limit in ms
Solution solutions = solver.findAllSolutions();
System.out.println(solutions);
```

Once the model is defined, the solver helps to define strategies, time limits, and to find the solutions. Many state-of-the-art constraints and strategies are already implemented in `choco-solver` and can be used directly. For users who want to use a specific search strategy, `choco-solver` allows to implement new strategies and use them during the solving. Users can also design and implement their own propagation algorithms, and attach them to the constraint propagation of `choco-solver`.

4. `https://pypi.org/project/pychoco/`

2.5 The Satisfiability Problem (SAT)

A special case of constrained problem is when all the variables are Boolean, and constraints are propositional formulas. We call this a SAT (satisfiability) problem. It is one of the central problems in computer science, both theoretically and practically. It was the first problem to be shown NP-complete [167], and it was then used to show that many other problems are NP-complete [173] using reductions to SAT. We present SAT as a special case of CP, but in practice, SAT was introduced before CP, and a lot of design ideas in CP come from SAT ideas.

2.5.1 Definition

Compared to CP, SAT is defined on a simpler constraint language for Boolean variables, called propositional formulas as defined in logic.

Definition 8 (Propositional formula)**.** Given variables x_i, a propositional formula is defined recursively such that

- a variable is a propositional formula;
- given a propositional formula ϕ, its negation $\neg\phi$ is a propositional formula;
- given two propositional formulas ϕ_1 and ϕ_2, and a binary operator $\square \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ (respectively for the `and`, `or`, implication, equivalence, and `xor` constraints), then $\phi_1 \square \phi_2$ is a propositional formula.

Propositional formulas allow for a great expressiveness, but for SAT solvers, a simpler input format is preferable. This format is called Conjunctive Normal Form (CNF).

Definition 9 (Conjunctive Normal Form (CNF))**.** Given a variable x, we call a *literal* the formula x and $\neg x$. A clause C is a disjunction of literals, i.e. $C = l_1 \vee l_2 \vee ... \vee l_k$. A formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses, i.e. $\phi = C_1 \wedge C_2 \wedge ... \wedge C_m$.

In the CNF form, clauses are the constraints: all the clauses must be satisfied. For a clause to be satisfied, at least one literal must be true. Converting propositional formulas into CNF form is not an easy task. Classically, implications, equivalences and `xor` constraints are reformulated using only negations, $\wedge$ and $\vee$. Then, propagation of negation to literals, and distribution operations (of $\wedge$ over $\vee$ or the vice versa) can convert the formula into CNF. However, there are formulas (such as `xor` constraints $x_1 \oplus x_2 \oplus \ldots \oplus x_n$) that

require an exponential number of clauses to be represented as CNF [29] without increasing the number of variables. There are other transformations, such as the Tseitin transformation [61]. The Tseitin transformation introduces new variables (one per sub-formula) and new constraints linking these new variables (containing at most 3 variables per clause). This transformation has a size (number of variables and number of constraints) at most polynomially larger than the original formula.

The DIMACS format is a common textual representation for CNF. The first line has the form

```
p cnf [v] [c]
```

where `[v]` is the number of variables, and `[c]` is the number of clauses. Then there is a line for each clause, ending with a 0. Each clause is a list of integers. The variables are numbered from 1 to n. Given $1 \leq i \leq n$, i represents the variable x_i, and $-i$ represents its negation $\neg x_i$.

Example. *The DIMACS representation of the CNF* $(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$ *is*

```
p cnf 4 3
-1 2 -3 4 0
1 2 0
1 -2 -4 0
```

This textual representation can then be passed to a SAT solver.

2.5.2 Solver

The basic algorithm for solving SAT problems uses a backtrack-search similar to CP. One of the main algorithms is DPLL [14], which is very similar to the one presented in Algorithm 2.1, but adapted to Boolean variables and clauses. It uses a special propagation step, called Boolean constraint propagation (or unit propagation). It searches for a clause where all but one literal is instantiated to false, and instantiates it to true.

Example. *We consider the same example formula as the previous example of DIMACS representation:* $(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$*. We assume that in the first step,* x_1 *was instantiated to 0 (*$false$*). The formula can then be simplified to*

$$(1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (0 \vee x_2) \wedge (0 \vee \neg x_2 \vee \neg x_4)$$

The first clause contains a true literal, it is then satisfied and can be forgotten. The literals set to 0 can be omitted from the clauses. This results in the following reduced formula:

$$(x_2) \wedge (\neg x_2 \vee \neg x_4)$$

In this formula, the first clause contains a single literal x_2, which must be set it to true (i.e. instantiated to 1). The formula is then reduced to only the clause (x_4), so x_4 must also be set to 1. After that, there is no more constraint, so x_3 can take any value.

As with CP, there are also search strategies for choosing the variable and value to branch on. Some strategies use the number of occurrences of the literal in the clauses to choose, such as `DLCS` [56]. Other select literals appearing in previous conflicting clauses, such as `VSIDS` [38].

DPLL was later improved by CDCL [57] (conflict-driven clause learning). In CDCL, when the search reaches a conflict, a new clause is learned to prevent the search from reaching the same conflict again. Information stored during unit propagation can be used to learn a new clause representing the conflict. Non-chronological backtracking can also be performed to backtrack from multiple levels at once.

SAT solvers use many other improvements that are beyond the scope of this thesis. For a detailed presentation of SAT solving, we refer the reader to the Handbook of Satisfiability [9].

2.6 Diversity

As mentioned earlier, solving a problem means finding solutions. However, users want to be given choices, and not just a single solution. This is even more important in the modelling phase, where the constraints may not all be formulated. On the other hand, showing the user all the solutions is overwhelming. On some problems there are too many solutions. On some instances (of software product lines) presented in Chapter 7 there are more than 10^{120} solutions. From all these possible solutions, an *interesting* small subset has to be extracted. Diversity is used here to formally define interestingness measures.

2.6.1 Definitions

How to evaluate the "interestingness" of a set of solutions ? From a user's point of view, some properties are desirable.

- Are the solutions far apart ? Solutions that are close to each other may contain redundancy and give little new information to the user.
- Do the solutions represent the other solutions well ? Solutions should be representative of all the options available to the user.
- Does the solution set have some coverage properties ? If the set of solutions offers guarantees (for example for software testing), the users can trust them.

The first item raises the question of *diversity*, as defined in [24]. The second item uses the notion of *representativeness*, as defined in [157]. The third item evaluates the guarantees of the solutions, as presented for example in Chapter 7 on the t-wise coverage of a test suite.

Distances

When referring to solutions as close or far from each other, an underlying distance is used. Many distances can be defined over solutions, depending on the application. Here, we define different useful distances.

Definition 10 (Distances)**.** We suppose that we have two solutions $s = (s_1, \ldots, s_n)$ and $s' = (s'_1, \ldots, s'_n)$, defined over numerical spaces (such as $\mathbb{R}$ or $\mathbb{Z}$). We define the following distances.

- Hamming, or l_0 distance, noted $\delta_{\mathcal{H}}$:

$$\delta_{\mathcal{H}}(s, s') = \sum_{i=1}^{n} \mathbb{1}_{s_i \neq s'_i}$$

  This distance is important when the dimensions of the solution do not represent integers that are meant to be compared. This is the case,for example, in configuration problems, where a variable may take the value $\{car, bus, plane\}$, but the reformulation to solve the problem identifies the identifiers with integers, for example with $\{car \mapsto 0, bus \mapsto 1, plane \mapsto 2\}$. Then the variable has no meaning in the integers, the only comparison possible to do is the equality (here, disequality) test. The Hamming distance can deal with these variables in a meaningful way.
- Manhattan, or l_1 distance, noted δ_{l_1}:

$$\delta_{l_1}(s, s') = \sum_{i=1}^{n} |s_i - s'_i|$$

This distance, unlike the Hamming distance, gives more leeway when comparing two solutions. It allows the values to be used to have a more precise comparison.

- Euclidean, or l_2 distance, noted δ_{l_2}:

$$\delta_{l_2}(s, s') = \sqrt{\sum_{i=1}^{n}(s_i - s'_i)^2}$$

This distance is the natural distance between two solutions in the space. For applications where the variables represent points in the space, it may be more appropriate to use the Euclidean distance.

Remark. *The three distances presented are classical when working in high-dimensional spaces, but other distances can be defined:*

- *These three distances are particular cases of the Minkowski distance, also called l_p distance, noted δ_{l_p} ($p = 0$ for Hamming, $p = 1$ for Manhattan, and $p = 2$ for the Euclidean distance). Given $p \geq 0$, the Minkowski distance is defined as*

$$\delta_{l_p}(s, s') = \left(\sum_{i=1}^{n}(s_i - s'_i)^p\right)^{1/p}$$

- *To use only integers, it is sometimes possible to use the squared Euclidean distance (i.e. omitting the square root).*
- *A combination of the above distances can be used. For example, the Hamming distance can be used for some dimensions, and the Euclidean distance for others.*
- *When the solutions are sets, other metrics can be used, such as the F-score, or the Jaccard index (presented and used in Chapter 7).*

When comparing or summing values, normalisation factors should be considered. Instead of giving each dimension the same weight, a distance can multiply the different dimensions by certain factors. This can either normalise the problem, or give more weight to certain dimensions that the user is more interested in.

The definition of distances heavily depends on the needs of the users. Some dimensions may be irrelevant, or important to them. In the following, we try to be as generic as possible, and use a distance noted δ.

Diversity Problems

In this section, we focus on satisfaction problems (CSPs). Defining diversity in an optimisation framework is harder. It is possible to transform an optimisation problem into a satisfaction problem by bounding the objective function (to be close to the optimal value), as done in Chapter 5. In Chapter 9 we study and define diversity in a multi-objective setting.

In satisfaction problems, diversity can be defined in a number of ways. In CP, it was introduced in [24]. The most general problem is `MaxDiverseKSet`. It searches for the most diverse set of k solutions of the problem.

Definition 11 (`MaxDiverseKSet`)**.** Let $k \geq 2$ be an integer and $\mathcal{P}$ be a CSP, with solutions $Sols(\mathcal{P})$, and δ be a distance over these solutions. `MaxDiverseKSet`(k) is the problem of finding a subset of solutions $\widetilde{S} \subseteq Sols(\mathcal{P})$ of size k that maximises the distances between the solutions, i.e.

$$\widetilde{S} = \underset{\substack{S \subseteq Sols(\mathcal{P}) \\ |S|=k}}{\operatorname{argmax}} \min_{\substack{s,s' \in S \\ s \neq s'}} \delta(s, s') .$$

In this definition, the solution set of the problem is the set that maximises the minimum distance between solutions. This ensures that all selected solutions are distant.

Remark. *In this definition, the minimum is used to aggregate all the pairwise distances. It is possible to use other aggregators to have a single value as an interestingness evaluator. The sum of the distances can also be used to aggregate all the pairwise distances. This is equivalent to averaging the pairwise distances. Using the sum aggregator has an impact on the resulting solution set. This impact is studied in Chapter 8.*

To understand the difficulty of this problem, we can look at the naive implementation. Given a problem $\mathcal{P}$, and a desired number of solutions k, we search for a subset S of $Sols(\mathcal{P})$ of size k. There are $\binom{|Sols(\mathcal{P})|}{k}$ such sets. In addition, it may already be difficult to find solutions of $\mathcal{P}$.

To ease this problem, we can search for the solutions one by one. The `MostDistant` problem searches for the most distant solution from a set of previously found solutions.

Definition 12 (`MostDistant`)**.** Let $S \in Sols(\mathcal{P})$ be a set of solutions and δ be a distance. `MostDistant`(S) is the problem of finding the solution $\widetilde{s}$ that is most distant from all the solutions in S, i.e. for all $s \in Sols(\mathcal{P})$,

$$\widetilde{s} = \underset{s \in Sols(\mathcal{P})}{\operatorname{argmax}} \min_{s' \in S} \delta(s, s') .$$

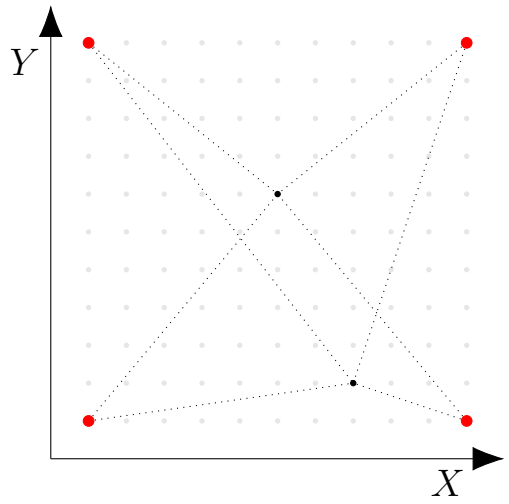


Figure 2.3 – Example of `MostDistant` solving. The variables are $X, Y \in \{0, 10\}$, $S = \{(0, 0), (10, 0), (10, 10), (0, 10)\}$

Example. *We show an example of* `MostDistant` *solution. We consider an unconstrained problem with variables X and Y with domains $\{0, \ldots, 10\}$, a set $S = \{(0, 0), (10, 0), (10, 10), (0, 10)\}$ of solutions already returned, and the Manhattan distance. Figure 2.3 shows a graphical representation of this problem, with the solutions in S marked in red. We search for the most distant point from S. Let us consider the point $(X, Y) = (7, 3)$. The minimum distance to the points in S is 4, with the point $(10, 0) \in S$. If we consider the point $(5, 6)$ as a candidate, the minimum distance to the points in S is 9, with the point $(0, 10)$ (and $(10, 10)$). The point $(5, 6)$ is more distant than $(7, 3)$ to the points in S. However, the* most *distant point is $(5, 5)$.*

The `MostDistant` problem can be seen as a greedy solving of `MaxDiverseKSet`. The solutions are drawn incrementally, and chosen to be the most distant from all the previously found solutions.

Remark. *For the diversity to be a meaningful question, the initial problem should have plenty of solutions. On a problem with very few solutions, all the solutions can be presented to the user.*

These two problems ensure that the set of solutions generated contains distant solutions. However, there is no information about the remaining (not returned) solutions of the problem. It would be interesting for a user to know if the returned solutions cover the whole set of solutions. This is the notion of representativeness. A returned solution *represents* the solutions that are the closest to it.

Definition 13 (`MostRepresentativeKSet` [157])**.** Let $k \geq 2$ be an integer and $\mathcal{P}$ be a CSP, with solutions $Sols(\mathcal{P})$, and δ be a distance over these solutions. `MostRepresentativeKSet`(k) is the problem of finding a subset of solutions $\widetilde{S} \subseteq Sols(\mathcal{P})$ of size k which minimises the diameter of the represented solution sets, i.e.

$$\widetilde{S} = \underset{\substack{S \subseteq Sols(\mathcal{P}) \\ |S|=k}}{\operatorname{argmin}} \max_{s \in Sols(\mathcal{P})} \min_{s' \in S} \delta(s, s') \,.$$

This definition is the same as `MaxDiverseKSet`, except for the evaluation of the solution sets. We describe this definition in more detail. The innermost min computes the minimum distance between a solution s of the problem $\mathcal{P}$ and the selected solutions (in S). This is equivalent to assigning the solutions to their closest representative solutions in S. Then the max computes from all the solutions of the problem, what is the largest distance to one of its representative solutions in S. The outer argmin searches for the set of representative solutions S that minimises the maximum distance to the representative solutions.

Remark. *The maximum distance to the representative solution is called the diameter. This definition tries to minimise the diameter of the representative solution set. It would also be interesting to count how many solutions are represented by each returned solution. Ideally, a user would like for each returned solution to represent the same number of solutions.*

The user may only want a *good* set of solutions, not necessarily the best one. The following section shows how CP can be used to generate sets of solutions (and approximations) for the problems we have presented.

2.6.2 Finding Diverse Sets

This section first presents COPs to solve exactly the `MaxDiverseKSet` and `MostDistant` problems. Then the PostHoc approach is presented. This is an approximation using a post-processing of the solutions.

Problem Reformulation

COPs for solving the `MaxDiverseKSet` and `MostDistant` problems were presented in [24]. Both assume that the original problem was already represented by a CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, with $\mathcal{X} = \{X_1, \ldots, X_n\}$ and $\mathcal{C} = \{C_1, \ldots, C_m\}$.

Remark. *The ability to find diverse solutions greatly depends on the ability to find solutions in the initial problem. In problems where finding a single solution is already very hard (takes hours or days of running time), finding diverse solutions may not be tractable. Diversity may more often be used in problems where solutions can easily be found, but there are many solutions in different parts of the search space.*

`MaxDiverseKSet` For the problem `MaxDiverseKSet`, a set of k solutions is wanted. To represent this problem as an optimisation problem, the initial model is duplicated k times. Each copy is one of the k solutions of the solution set. Formally, we define new variables (we name the "new" variables and constraints using exponents):

$$\forall 1 \leq i \leq k, \text{ we define the variables } X_1^i, \ldots, X_n^i \,.$$

The domains $\mathcal{D}'$ of these new variables are duplicated from the domains of the initial model:

$$\forall 1 \leq i \leq k, \forall 1 \leq j \leq n, \mathcal{D}'(X_j^i) = \mathcal{D}(X_j) \,.$$

The constraints are also duplicated. For $1 \leq i \leq k$, for each constraint $C_j \in \mathcal{C}$, a new constraint C_j^i is created such that $scp(C_j^i) = (X^i)_{X \in scp(C)}$ and $rel(C_j^i) = rel(C_j)$.

The problem defined this way is simply the initial model duplicated k times. We now need to link the different duplicated models with distance constraints. We define new variables to store the pairwise distances between solutions.

$$\forall 1 \leq i < j \leq k, \text{ we define the variables } d_{i,j} \,.$$

We do not specify the domain of these variables, as it depends on the distance used, and the domains of the variables of the initial model. We can now add the constraints on the distances:

$$\forall 1 \leq i < j \leq k, \text{ we define the constraints } d_{i,j} = \delta((X^i)_{X \in \mathcal{X}}, (X^j)_{X \in \mathcal{X}}) \,.$$

We assume that we are able to define the distance δ using the constraints of the language. It may be necessary to use intermediate variables to do this. We can now define the objective variable as the minimum of all the distances:

$$obj = \min_{1 \leq i < j \leq k} d_{i,j} \,.$$

The complete optimisation problem is

$$\mathcal{P}' = \begin{array}{l} \langle \quad \bigcup_{1 \leq i \leq k} \{X_1^i, \ldots, X_n^i\}, \\ \quad \mathcal{D}', \\ \quad \bigcup_{1 \leq i \leq k} \{C_1^i, \ldots, C_m^i\}, \\ \quad obj \rangle \end{array}$$

Finding a solution to $\mathcal{P}'$ that maximises the variable *obj* gives an optimal solution set for the `MaxDiverseKSet` problem.

Example. *We show the same reformulation on a simple example. We consider the problem with only two variables X and Y with domains $\{0, \ldots, 2\}$ and the constraint $X + Y \leq 2$.*

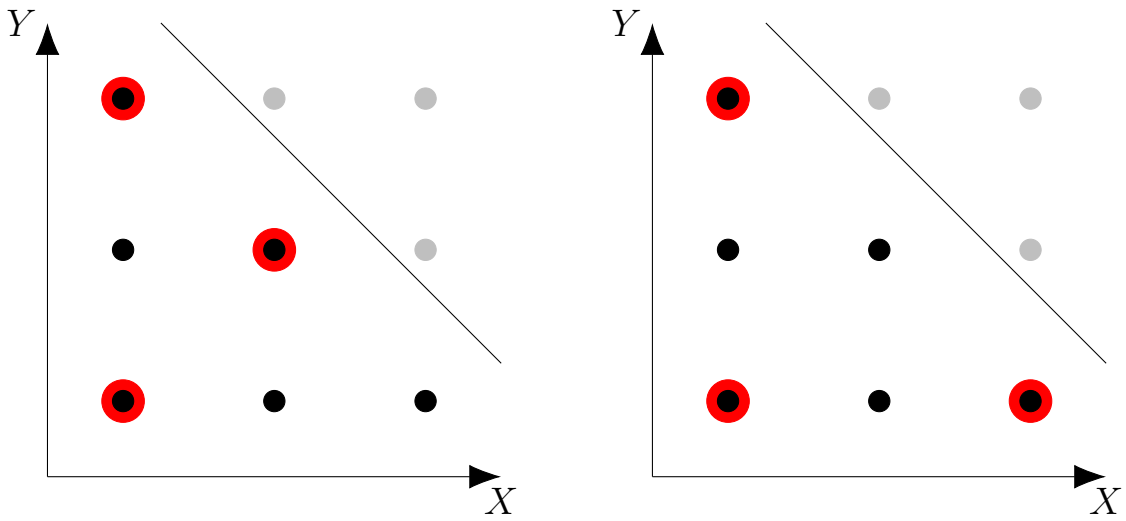
We want to find a solution set for the `MaxDiverseKSet`(3)*. We copy the initial problem 3 times with the variables X^1, X^2, X^3 and Y^1, Y^2, Y^3, all with domains $\{0, \ldots, 2\}$. We add the same constraints on these variables as in the initial problem: $X^1+Y^1 \leq 2$, $X^2+Y^2 < 3$, and $X^3+Y^3 \leq 2$. Here, we use the Manhattan distance δ_{l_1}. We create the distance variables $d_{1,2}, d_{1,3}, d_{2,3}$ with domains $\{0, \ldots, 4\}$ (we know that the maximum Hamming distance for two solutions of the problem is 4). We now add the constraints on the distances:*

$$\begin{aligned} d_{1,2} &= |X^1 - X^2| + |Y^1 - Y^2| \\ d_{1,3} &= |X^1 - X^3| + |Y^1 - Y^3| \\ d_{2,3} &= |X^2 - X^3| + |Y^2 - Y^3| \end{aligned}$$

We create the objective variable $obj =$ `min`$(d_{1,2}, d_{1,3}, d_{2,3})$*. This variable obj should be maximised.*

Figure 2.4 shows two optimal solution sets that can be generated using this model. The optimal minimum distance is 2. From a user's point of view, the second solution set (in Figure 2.4b) seems more diverse*, because two solutions have a distance of 4 (the top and rightmost solutions). However, this is not captured by the model, as only the* minimum *distance is used to evaluate the set. This behaviour of the* min *aggregator is studied in Chapter 8.*

This formulation of the problem as a COP allows finding the optimal set of solutions that maximises the minimum pairwise distance. However, copying the problem k times adds a level of complexity. Satisfying the constraints on each problem can be hard, so finding the k most optimal solutions is even harder.



(a) (b)

Figure 2.4 – Two examples of optimal diverse solution sets. The black dots are solutions, the grey dots are the search space, the diagonal line is the constraint $X + Y < 3$. The red circles are the solutions in the diverse solution set.

`MostDistant` To avoid having to solve a problem k times larger, it is possible to generate the solutions one by one. From an initial set $S = \emptyset$, the most distant solution from the set S is found iteratively and added to it. Solving the problem `MostDistant`(S) k times produces an approximation of `MaxDiverseKSet`(k). Given a problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, it is possible to reformulate $\mathcal{P}$ to find the most distant solution from those in the set S (i.e. to solve the `MostDistant`(S) problem). This reformulation is simpler than the one for `MaxDiverseKSet` because only one solution is searched. For each solution $s \in S$ we create a variable d_s for the distance between the new solution and the previous ones. We add the constraints

$$\forall s \in S, d_s = \delta(\mathcal{X}, s)\,.$$

We then create the objective as before

$$obj = \min_{s \in S} d_s\,.$$

Finding the optimal solution for this COP gives the solution to the `MostDistant`(S) problem. Compared to the previous reformulation, this one is easier to solve, because the initial problem $\mathcal{P}$ is not duplicated. However, after k calls, the generated solution set is an approximation of the `MaxDiverseKSet`(k) problem. In Chapter 8 we show that calling

```
1 Function PostHoc(P, k, Find, K, Div)
      Data: P a problem, integers k and K such that K > k, an algorithm Find
            finding solutions of P, and an algorithm Div finding a diverse subset
            of solutions.
      Result: A set of k diverse solutions of P
2     S_K ← Find(P, K);                                  // find K solutions
3     S_k ← Div(S_K, k);                                 // extract k solutions
4     return S_k;
```

Algorithm 2.4: PostHoc: diverse solutions in two steps

k times `MostDistant` is a 2-approximation of the `MaxDiverseKSet` (i.e. in the worst case, the minimum distance of the solution set is half the optimal distance).

Post-Processing Solutions

Another approach, called PostHoc, was designed in [28] to approximate the `MaxDiverseKSet` problem. The idea is to split the `MaxDiverseKSet` problem into two problems: first finding solutions to the initial problem, and then finding a diverse subset of those solutions.

Algorithm 2.4 presents PostHoc. PostHoc generates k solutions. To do so, it first generates $K > k$ solutions using the Find function. Then it uses the Div function to extract k diverse solutions from the K initial ones. Using the two functions Find and Div allow to split the problem into two simpler sub-problems: finding solutions (to a possibly hard problem), and extracting diverse solutions.

For the Find function, an exact approach is not wanted (this is exactly what the PostHoc approach avoids). The Find function can be implemented using a greedy approach such as calling `MostDistant` K times. It can also use a random approach to generate the solutions (such as using the RandomSearch search strategy). A bad implementation would be using the default backtrack-search, because the solutions returned are close to each other. The implementation of the Find function needs to generate solutions in the whole solution space, but not necessarily diverse: this is what the second step is for.

The Div function extracts k diverse solutions from the K solutions generated by Find. When K and k are small enough, an exact approach can be implemented (searching among $\binom{K}{k}$ subsets of solutions). If this exact approach would take too much time, a greedy approximation (such as `MostDistant`, but with solutions already known) can be

implemented.

The choice of K depends on the desired quality of the resulting set. The more solutions generated, the more likely it is to find a good diverse subset. We have evaluated the PostHoc approach experimentally in Chapter 6.

	6	2				8		
						3		9
4			5				6	
	1				9			
	9		2	5	6		3	
			3				8	
	3				4			1
5		1						
		4				2	9	

Figure 2.5 – A *Sudoku* grid. When there is still place at the end of a chapter, I add a logic game that you can solve during a break.

Chapter 3

Probabilities

3.1 Introduction

Randomness is a term we use to describe behaviour that cannot be predicted. It starts with a simple coin toss. A fair coin will land heads half of the time on average. However, with practice, by always flipping the coin always in the same manner, it is possible to increase the odds of getting a chosen side. Throwing a dice works in the same way. If the initial conditions are the same, the outcome of the dice will always be the same. However, a small change in the initial conditions (an angle of the table, the presence of an air current) will completely change the result. This behaviour is chaotic. It is extremely difficult to study a chaotic system accurately with a deterministic approach: this is where randomness comes in. Instead of trying to analyse the behaviour perfectly, it is possible to analyse the average result. Albert Einstein's famous quote about quantum mechanics is another example of the use of randomness as a modelling tool: "God does not play dice with the universe." The random behaviour of quantum particles is a modelling approximation, because we do not know the exact underlying behaviour of such particles.

In this thesis, we use randomness to modify the behaviour of otherwise deterministic algorithms. The use of randomness in optimisation algorithms is not new. For example Simulated Annealing, Genetic Algorithms and Monte-Carlo Tree Search all use randomness and would not work without it. For example, in Monte-Carlo Tree Search, the future winner of a given game position cannot be accurately evaluated due to the combinatorial explosion of the states of a game. A random game is played, and under good conditions, doing enough of these random games gives a good evaluation of the game position.

The analysis of random algorithms should be done in a probabilistic framework. For example, it is rarely instructive to analyse the *worst case* of a random algorithm, as it is commonly done for deterministic algorithms, because this worst case may have an extremely low chance of happening. For random algorithms, other properties are important, such as the average number of operations, the average quality of the solutions (given a

metric), the probability of failure, the distribution of the solutions, and so on.

In this chapter we give the definitions and notations of the probability concepts used in the rest of this thesis. It facilitates the other chapters by defining and explaining all the probability concepts we use. It is not intended to be as complete as a textbook: we focus on the topics used in this thesis, for example, we mostly use finite and discrete probabilities.

The chapter is structured as follows: in Section 3.2 we define the basic notions of probabilities, and classical random distributions. In Section 3.3 we present properties of random algorithms (such as samplers). Finally, in Section 3.4 we present hashing constraints used in several samplers.

3.2 Definitions and Notations

We start by defining the basic concepts.

3.2.1 Probability

It is possible to define probabilities very formally in order to be as generic as possible. However, in this section we have opted for a more intuitive presentation. For a more detailed presentation of the foundations of probability theory (such as probability spaces or measures), we refer the reader to probability textbooks.

Definition 14. We define the basic notions of probabilities, and the notations we use in this thesis.

- A distribution is a function associating each element s of a finite set $\mathcal{S}$ to a probability p_s such that $\sum_{s\in\mathcal{S}} p_s = 1$.
- A random variable is a variable that takes a value in the set $\mathcal{S}$ under a given distribution.
- An event is a set of outcomes, i.e. given a random variable Y and a set $S \subseteq \mathcal{S}$, $Y \in S$ is an event.
- We note $\mathbb{P}(Y \in S)$ the probability of the event $Y \in S$. It is the sum of the probabilities of each element of S, i.e. $\mathbb{P}(Y \in S) = \sum_{s\in S} p_s$. When S contains a single element s, the event $Y \in S$ is also noted $Y = s$, and $\mathbb{P}(Y = s) = p_s$.
- Given two events A and B, the conditional probability of A knowing B is $\mathbb{P}(A \mid B) = \mathbb{P}(A \wedge B) / \mathbb{P}(B)$.

- Two events A and B are independent if $\mathbb{P}(A \mid B) = \mathbb{P}(A)$, or equivalently, $\mathbb{P}(A \wedge B) = \mathbb{P}(A)\,\mathbb{P}(B)$.
- If $\mathcal{S}$ is a finite subset of the integers $\mathbb{Z}$, and Y is a random variable on a distribution on $\mathcal{S}$, then the expected value of Y is $\mathbb{E}(Y) = \sum_{s \in S} s \cdot \mathbb{P}(Y = s)$.

Example. *We take the example of a 6-sided dice. A random variable Y recording the outcome of a roll takes values in the set $\{1, \ldots, 6\}$. The distribution of the values is uniform $\mathcal{U}(\{1, \ldots, 6\})$ such that $p_i = 1/6$ for $i \in \{1, \ldots 6\}$. The random event of getting an even value is $Y \in \{2, 4, 6\}$, and has probability $\mathbb{P}(Y \text{ is even}) = \mathbb{P}(Y \in \{2, 4, 6\}) = 1/2$. Given the events $A =$ "$Y = 2$" and $B =$ "Y is even", the probability of the roll being 2 knowing that it is an even number is $\mathbb{P}(A \mid B) = 1/3$. The expected value of the variable Y is 3.5 $(= \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6})$.*

The following property states that the expected value is linear, i.e. the expected value of the sum of the variables is the sum of the expected values. Note that there is no assumption about the variables (they do *not* have to be independent).

Property 1 (Linearity of Expected Value)**.** *Let $Y_1, \ldots, Y_n$ be random variables taking values in $\mathbb{R}$. Then,*

$$\mathbb{E}\left(\sum_{i=1}^{n} Y_i\right) = \sum_{i=1}^{n} \mathbb{E}(Y_i) \ .$$

In the following we define different distributions (Bernoulli, uniform and weighted), we define what a sampler is, and we introduce hashing constraints.

3.2.2 Distributions

Here we define different distributions.

Uniform Distribution When all the outcomes are equiprobable, the distribution is said to be uniform. It is denoted $\mathcal{U}(\mathcal{S})$, and $p_s = 1/|\mathcal{S}|$. In the pseudocode, we assume that we have access to a function $\textsc{Random}(\mathcal{S})$ which returns a random element according to the uniform distribution. We also assume that $\textsc{Random}()$ returns a random real number from $[0, 1]$ (randomness is harder to define on continuous spaces, we simply assume that we have access to such a function).

Weighted Distribution When one wants to specify the exact distribution, it is possible to use a weighted distribution. For each $s \in \mathcal{S}$ a weight ω_s is defined. The weighted distribution is $\mathcal{W}(\mathcal{S}, \omega)$ and the associated probabilities are $p_s = \omega_s / \sum_{s' \in \mathcal{S}} w_{s'}$. Generating a random variable from a weighted distribution is easy when the weights and the set $\mathcal{S}$ are known [192], and a uniform generater is avaialable.

Bernoulli Distribution The Bernoulli distribution is a distribution on a set of two elements, usually $\{0, 1\}$. The probability of getting the value 1 is noted p, and the distribution is noted $\mathcal{B}(p)$. Let Y be a random variable following a Bernoulli distribution, then $\mathbb{E}(Y) = \mathbb{P}(Y = 1) = p$.

Implementation of Random Numbers In this thesis, I use the `Java` programming language. The random number generator we use is the default one in Java: `java.util.Random`. This generator uses a formula of linear congruence to modify a 48-bits seed, given as input. The Java documentation refers to [175] section 3.2.1 for more information. This randomness generator has flaws (notably a period of 2^{48}), but is sufficient for our needs (as shown in [163]). The seeds allow us to have deterministic executions of the code (i.e., two different executions using the same seed will use the same random numbers).

3.3 Random Algorithms

Using randomness in algorithms helps to make an algorithm non-deterministic, and thus returning solutions in another order. However, it is important to know the properties of new algorithm.

3.3.1 Samplers

A sampler is an algorithm that randomly generates solutions to a problem. When designing a sampler, we are interested in the distribution of the solutions guaranteed by the sampler. The most common guarantee is the uniformity of the sampling.

Definition 15 (Uniform Sampler)**.** Given an input problem $\mathcal{P}$, an algorithm $\mathcal{U}$ is a *uniform sampler* iff

$$\forall s \in Sols(\mathcal{P}), \mathbb{P}(\mathcal{U}(\mathcal{P}) = s) = \frac{1}{|Sols(\mathcal{P})|}.$$

Remark. *We want to point out that in this definition, $\mathcal{U}$ is not a function in the mathematical sense because it returns different outputs (random solutions) given the same input (the problem).*

Sometimes, it is difficult to guarantee the uniformity of the sampling. There are relaxations of the definition of a uniform sampler to allow for approximate uniform sampling.

Definition 16 (Approximately Uniform Sampler)**.** Given an input problem $\mathcal{P}$, an algorithm $\mathcal{U}$ is an *approximately* uniform sampler iff there exists $\epsilon > 0$ such that

$$\forall s \in Sols\left(\mathcal{P}\right), \frac{1}{\left(1+\epsilon\right)\left|Sols\left(\mathcal{P}\right)\right|} \leq \mathbb{P}\left(\mathcal{U}(\mathcal{P}) = s\right) \leq \frac{1+\epsilon}{\left|Sols\left(\mathcal{P}\right)\right|}.$$

Remark. *This definition is the multiplicative-approximate uniform sampler: bounding the probability between $\frac{1}{1+\epsilon}$ and $1+\epsilon$. Another definition, the additive-approximate uniform sampler, bounds the probability between $1-\epsilon$ and $1+\epsilon$. The additive-approximate definition is more relaxed than the multiplicative one, because $1-\epsilon \leq \frac{1}{1+\epsilon}$.*

An approximate sampler guarantees that the distribution of the solutions is close (up to ϵ) to the uniform distribution. In some samplers, ϵ can be given as a parameter. If ϵ is close to 0, the sampling is close to uniform, but more computation may be required by the sampler to give this guarantee.

Another guarantee, more relaxed than the approximate uniformity, is the near-uniformity.

Definition 17 (Near-Uniform Sampler)**.** Given an input problem $\mathcal{P}$ and an approximation factor $0 < c < 1$, an algorithm $\mathcal{U}$ is a *near*-uniform sampler iff

$$\forall s \in Sols\left(\mathcal{P}\right), \mathbb{P}\left(\mathcal{U}(\mathcal{P}) = s\right) \geq \frac{c}{\left|Sols\left(\mathcal{P}\right)\right|}.$$

The near-uniformity guarantees that all the solutions have at least a fixed probability of being sampled. As c tends to 1, the sampling tends to uniformity.

Chapter 4 presents several samplers of SAT and CP problems. Some samplers have guarantees, and some focus on efficiency. This gives the users a choice depending on their application.

3.3.2 Other properties

It is important to know whether a random algorithm terminates. Algorithms that terminate in a random running time are called Las Vegas algorithms. Algorithms that

terminate in a deterministic time, but that can produce a wrong answer (with a bounded probability) are called Monte-Carlo algorithms.

Probably Approximately Correct Algorithm

Monte-Carlo algorithms can have guarantees: the guarantee that they will not fail, and the guarantee that they will give an answer close to the actual solution. This can be formalised by the *probably approximate* property. We define it for approximate model counting algorithms, i.e. counting the number of solutions of a model.

Definition 18 (Probably Approximately Correct)**.** Given a problem $\mathcal{P}$, a tolerance $\epsilon > 0$ and a confidence δ, a *probably approximately correct* (PAC) model counter $\mathcal{A}$ will with probability at least δ give an answer that is close (up to a factor ϵ) to the actual solution $|Sols(\mathcal{P})|$, i.e.

$$\mathbb{P}\left(\frac{|Sols(\mathcal{P})|}{1+\epsilon} \leq \mathcal{A}(\mathcal{P}) \leq |Sols(\mathcal{P})|(1+\epsilon)\right) \geq \delta\,.$$

Chapter 4, Section 4.2.1 shows the links between approximate counting and sampling.

Randomness as a Constraint

In this thesis, we consider randomness as a global property, i.e. a statistical property on a subset of solutions. Formally, it does not make sense to say that one solution is *more random* than another solution. However, when showing a solution to a user, the *gambler's fallacy* can bias their opinion about the randomness of the solution. For example, although the sequences 1010110001 and 1111111111 are two equiprobable outcomes of a uniform sampler over $\{0,1\}^{10}$, the former sequence *seems more random* than the latter. In [45], the authors formalise this intuition using Kolmogorov complexity. However, since Kolmogorov complexity is uncomputable, the authors approximate the *entropy* of a sequence (i.e. a solution of a problem) by using compression algorithms. If a compression algorithm cannot compress a sequence much, this intuitively means that it has a random-like behaviour. The authors design two constraints based on the main compression algorithms: a *frequency entropy constraint* which limits the number of occurrences of values, and a *dictionary entropy constraint* which limits the number of occurrences of k-grams (blocks of k adjacent symbols). The generated solutions are sequences that cannot be be compressed, i.e. sequences that will appear random to a user.

Stochastic Constraint Programming

In some applications, uncertainty should be taken into account in the modelling phase. For example in, production planning, demand may vary from month to month. It can be modelled as a stochastic variable: a variable that takes a random value. Then, the goal is to find a schedule that satisfies the demand with a certain probability.

Stochastic Constraint Programming [60, 62] is an extension of CP to allow the modelling of uncertainty. In addition to the usual decision variables, stochastic variables are introduced to model probabilistic behaviour. Instead of a fixed domain, these variables follow a probability distribution. Solving the stochastic CSP means finding an assignment to the decision variables such that the probability that the constraints are satisfied is greater than a threshold θ chosen by the user. In [62], the author proposes an adaptation of the classical CP backtrack-search algorithm to account for the stochastic variables and in [59, 60], the authors show how to find solutions to the stochastic CSP by solving classical CSPs. Recently, in [40], stochastic constraint programming has been extended with *distribution variables*. The domain of these variables are probability distributions, so that during the solving, these variables are instantiated to a distribution, and then they act as stochastic variables.

3.4 Hashing Constraints

In this section, we present hashing constraints, which are an adaptation of hash functions to constrained problems. Hash functions are a powerful tool, mostly studied for the hash table data structure. In hash tables, the number of operations depends on the number of collisions. To reduce the number of collisions, powerful hash functions are designed. However, if the hash function is selected beforehand, a worst case can be designed to have many collisions. To avoid this worst case, hash functions are picked randomly from a family of functions. For well constructed families of hash functions, the average number of operations is constant (amortized).

Hashing constraints provide the same framework as hash functions but for constrained problems. When added to a model, hashing constraints act in the same way as other constraints, reducing the search and solution space. However, to ensure some properties on which part of the solution space is removed, they are randomly generated.

Let $\mathcal{P}$ be a problem with n variables $X_1, \ldots, X_n$. Let $\mathcal{H}$ be a family (i.e. a set) of constraints on the variables of $\mathcal{P}$. Let h be a random constraint of $\mathcal{H}$. h can be seen as a random

variable taking a constraint as a value. Then the (random) constraint $h(X_1, \ldots, X_n)$ can be added to the problem $\mathcal{P}$ to reduce the number of solutions.

When a hashing constraint is added to a problem, it is chosen randomly from a family of hashing constraints. A family of hashing constraint should not favour any solution, so all the solutions in the solution space should have the same probability of satisfying the hashing constraint.

Definition 19 (Uniform Partitioning)**.** Let $\mathcal{X}$, and let $\mathcal{H}$ be a family of constraints on all the variables of $\mathcal{X}$. The family $\mathcal{H}$ *uniformly partitions* the space iff there exists a constant c such that for a random $h \in \mathcal{H}$ and for all instantiations σ,

$$\mathbb{P}\left(\sigma \in rel(h)\right) = 1/c$$

The uniform partitioning of the search space ensures that the hashing constraints reduce the search space **and** the solution space. It should be remarked that the constant c does not depend on the other constraints imposed on the variables. This is one of the powers of hashing constraints: they are oblivious to the other constraints of the problem.

Another important property when designing a family of hashing constraints is the r-independence of the family.

Definition 20 (r-independence)**.** Let $\mathcal{X}$, and let $\mathcal{H}$ be a family of constraints on all the variables of $\mathcal{X}$. The family $\mathcal{H}$ is r-independent iff for a random $h \in \mathcal{H}$ and for $\sigma_1, \ldots \sigma_r$ instantiations of the variables in $\mathcal{X}$

$$\mathbb{P}\left(\sigma_r \in rel(h) \mid \sigma_1 \in rel(h) \wedge \ldots \wedge \sigma_{r-1} \in rel(h)\right) = \mathbb{P}\left(\sigma_r \in rel(h)\right)$$

Intuitively, the r-independence means that knowing that $r-1$ instantiations satisfy h gives no information about any r-th instantiation. This means that the solution space of the problem is partitioned independently of the solutions. This property is very important: if the hashing constraint were not independent, some sets of solutions would never be partitioned (and would always be together in the hashed space). The r-independence is a very hard property to guarantee.

We now present two families of hashing constraints.

3.4.1 XOR constraints

XOR constraints, introduced in [79], are a 3-independent family of hashing constraints designed for Boolean problems.

Definition 21. Given n Boolean variables $x_1, \ldots, x_n$ and Boolean coefficients $a_i \in \{0, 1\}$ for $1 \leq i \leq n+1$, an XOR constraint is

$$a_1 x_1 \oplus \ldots \oplus a_n x_n = a_{n+1}$$

The family of all XOR constraints on n variables is noted $\mathcal{H}_{xor}(n)$, i.e.

$$\mathcal{H}_{xor}(n) = \{a_1 x_1 \oplus \ldots \oplus a_n x_n = a_{n+1} \mid a_1, \ldots, a_{n+1} \in \{0, 1\}\} .$$

Notation. *We also define the family of systems of m XOR constraints on n variables $\mathcal{H}_{xor}(n, m)$:*

$$\mathcal{H}_{xor}(n, m) = \{h_1 \wedge \ldots \wedge h_m \mid h_1, \ldots, h_m \in \mathcal{H}_{xor}(n)\} .$$

This is equivalent to adding m XOR constraints to a problem.

To ensure the properties of the family of hashing constraint, the hashing constraint should be picked randomly and uniformly from the family. Picking a random XOR constraint (from $\mathcal{H}_{xor}(n)$) can be done by randomly generating the coefficients $a_1, \ldots a_{n+1}$. Picking a random system of m XOR constraints (from $\mathcal{H}_{xor}(n, m)$) can be done by either generating m random XOR constraints, or directly by generating a matrix of $(n+1) \times m$ coefficients.

Property 2 (Uniform Partitioning [79])**.** *Let $x_1, \ldots, x_n$ be n variables, and σ be an instantiation on these n variables. Let h be a random XOR constraint from $\mathcal{H}_{xor}(n)$, then*

$$\mathbb{P}(\sigma \in rel(h)) = 1/2 .$$

Let H be a random system of m XOR constraints from $\mathcal{H}_{xor}(n, m)$, then

$$\mathbb{P}(\sigma \in rel(H)) = 1/2^m$$

This property means that on average, an XOR constraint reduces the search and solution space by a factor 2. A system of m XOR constraints divides the search and solution space by a factor 2^m. Moreover, the family of XOR constraints is 3-independent.

Property 3 (3-independence [79])**.** *The family* $\mathcal{H}_{xor}(n)$ *is* 3*-independent.*

This property ensures that there is no dependency between any 3 solutions when hashing with XOR constraints. It is at the heart of the proofs of uniformity (or almost uniformity) of samplers. Samplers using the $\mathcal{H}_{xor}(n, m)$ family of hashing constraints are presented in Chapter 4 Section 4.2.1.

3.4.2 Linear Modular Equality System

The family $\mathcal{H}_{xor}(n)$ of XOR constraints is defined over Boolean variables. In CP, the variables have a larger domain. Linear modular equalities extend the family $\mathcal{H}_{xor}(n)$ to integer variables.

Notation. *Let* p *be a prime number. We define* $\mathbb{F}_p$ *to be the finite field of elements* $\{0, \ldots, p-1\}$ *with operations modulo* p*. For example, the Boolean field is* $\mathbb{F}_2$*, where the addition (*`xor`*) and multiplication (*`and`*) are done modulo* 2*. The term* field *means that the addition and multiplication have good properties on* $\mathbb{F}_p$*. In particular, every element (except 0) has an inverse, which means that the division is also well defined.*

Family of Constraints

Linear modular equalities as hashing constraints were introduced in [41]. They are extensions of the XOR constraints (linear constraints in $\mathbb{F}_2$) to integers (in $\mathbb{F}_p$). For a more detailed analysis, and more theoretical background we refer the reader to the original article.

Definition 22 (Linear Modular Equality)**.** Let p be a prime number. Let $a_1, \ldots, a_n \in \mathbb{F}_p$ be coefficients, $b \in \mathbb{F}_p$ be a constant, and $X_1, \ldots, X_n$ be n variables. A *linear modular equality* is an equality

$$\sum_{i=1}^{n} a_i X_i \equiv b \pmod{p}.$$

The family $\mathcal{H}_{(\text{mod } p)}(n)$ is the set of all linear modular equalities modulo p, i.e.

$$\mathcal{H}_{(\text{mod } p)}(n) = \left\{ \sum_{i=1}^{n} a_i X_i \equiv b \pmod{p} \mid a_1, \ldots, a_n \in \mathbb{F}_p, b \in \mathbb{F}_p \right\}$$

XOR constraints divided the space by a factor 2, and similarly, linear modular equalities divide the space by a factor p.

Property 4 (Uniform Partitioning [41]). *Let $X_1, \ldots, X_n$ be n variables, and σ be an instantiation on these n variables. Let h be a random linear modular equality constraint from $\mathcal{H}_{(\text{mod } p)}(n)$, then*

$$\mathbb{P}(\sigma \in rel(h)) = 1/p\,.$$

The prime number p must to be chosen to be larger than the range of the domains of the variables to ensure an independent hashing of the space. With such a prime number p, these linear modular equalities can be used as hashing constraints.

Theorem 1 (2-independence [41]). *Let $X_1, \ldots, X_n$ be integer variables, and let p be a prime number greater than the range of the domains (i.e. $p > \max_{1 \leq i \leq n}(\max \mathcal{D}(X_i) - \min \mathcal{D}(X_i))$). Then the family $\mathcal{H}_{(\text{mod } p)}(n)$ of linear modular hashing constraints is 2-independent.*

The prime number p has to be bigger than the maximum range of the domains because all the values of the domains should be mapped to a different value modulo p. For example, if a value v and $v + p$ are in the domain of a variable, then modulo p, both values are mapped to the same integer.

To use these hashing constraints, one must be able to randomly pick in $\mathcal{H}_{(\text{mod } p)}(n)$. This can be achieved by randomly picking the coefficients a_i and the constant b in $\mathbb{F}_p$.

Often, several hashing constraints are added to a problem. When using linear modular equalities (with a fixed p), all the constraints can be combined into a single system of modular equalities. We now present the filtering algorithm for such a system.

Propagation Algorithm

The algorithm used to propagate inconsistent values is based on the Gauss-Jordan elimination. $\mathbb{F}_p$ is a field when p is a prime number, so it is possible to apply the Gauss-Jordan elimination to a system of equations modulo a prime number.

The first step of the algorithm is to put the system into row reduced echelon form using the Gauss-Jordan elimination. This step splits the variables into two sets: the parametric variables and the non-parametric variables. The property of this form is that if all the parametric variables are instantiated, then only one value is allowed for the non-parametric variables.

No propagation is done at the beginning of the search, when only a few variables are instantiated. Propagation starts when the size of the Cartesian product of the domain of

the parametric variables is less than 1000 (i.e. when $|\prod_{X \in \mathcal{X}} \mathcal{D}(X)| \leq 1000$). The threshold 1000 was chosen experimentally by the authors as a good compromise between propagation being too late (if a small threshold is chosen) and the enumeration of values being too large (if a large threshold is chosen). Once all the possible values for the parametric variables have been enumerated, it is possible to compute the corresponding values for the non-parametric variables. This enumeration is exactly the set of instantiations allowed by the constraints (at this particular step of the search with many variables already instantiated). These instantiations serve as a support for the application of a filtering algorithm such as table constraints.

Example Below we show how the propagator works on an example. We consider a problem with four variables X_1, X_2, X_3, and X_4 with domains $\mathcal{D}(X_1) = \mathcal{D}(X_3) = \{0, \ldots, 4\}$, $\mathcal{D}(X_2) = \{0, 1, 2\}$, and $\mathcal{D}(X_4) = \{0, 1\}$. We consider a randomly generated system of linear modular equality constraints. To generate this system, the coefficients and the constant are randomly picked in $\mathbb{F}_5 = \{0, \ldots, 4\}$ (i.e. random hashing constraints in $\mathcal{H}_{(\text{mod } 5)}(4)$). This yields a system like the following:

$$\begin{cases} 3X_1 & +2X_2 & +3X_3 & +1X_4 & \equiv 4 \pmod 5 \\ 4X_1 & +1X_2 & +1X_3 & +0X_4 & \equiv 0 \pmod 5 \end{cases}$$

- The first step is to apply Gauss-Jordan elimination. The exact operations on the lines are $L_1 \leftarrow 2L_1; L_2 \leftarrow L_2 - 4L_1; L_2 \leftarrow 3L_2; L_1 \leftarrow L_1 - L_2$. This leads to the following system of equations:

$$\begin{cases} 1X_1 & +4X_2 & +0X_3 & +1X_4 & \equiv 0 \pmod 5 \\ 0X_1 & +0X_2 & +1X_3 & +1X_4 & \equiv 3 \pmod 5 \end{cases}$$

- We can now identify the parametric and the non-parametric variables. The non-parametric variables are X_1 and X_3 (used in the pivot operation of the Gauss-Jordan elimination). The parametric variables are X_2 and X_4. We can rewrite the system to make this clear:

$$\begin{cases} X_1 & \equiv 0 & -4X_2 & -1X_4 \pmod 5 \\ X_3 & \equiv 3 & -0X_2 & -1X_4 \pmod 5 \end{cases}$$

- Now, by definition of parametric variables, fixing values for X_2 and X_4 will fix the

values of X_1 and X_3. By enumerating the domains of X_2 and X_4 we can enumerate all the possible instantiations that satisfy the constraint:

$$\begin{aligned}(X_1, X_2, X_3, X_4) \in \{ \quad & (0,0,3,0),\\ & (4,0,2,1),\\ & (1,1,3,0),\\ & (0,1,2,1),\\ & (2,2,3,0),\\ & (1,2,2,1)\}\end{aligned}$$

- The constraint is converted to a table constraint using these instantiations.

In this example, the number of enumerated tuples was less than 1000, but in practice this will not be the case at the start of the search. In this case, the propagator waits until the threshold is reached (either other propagators or decisions during the search will reduce the domain of some parametric variables).

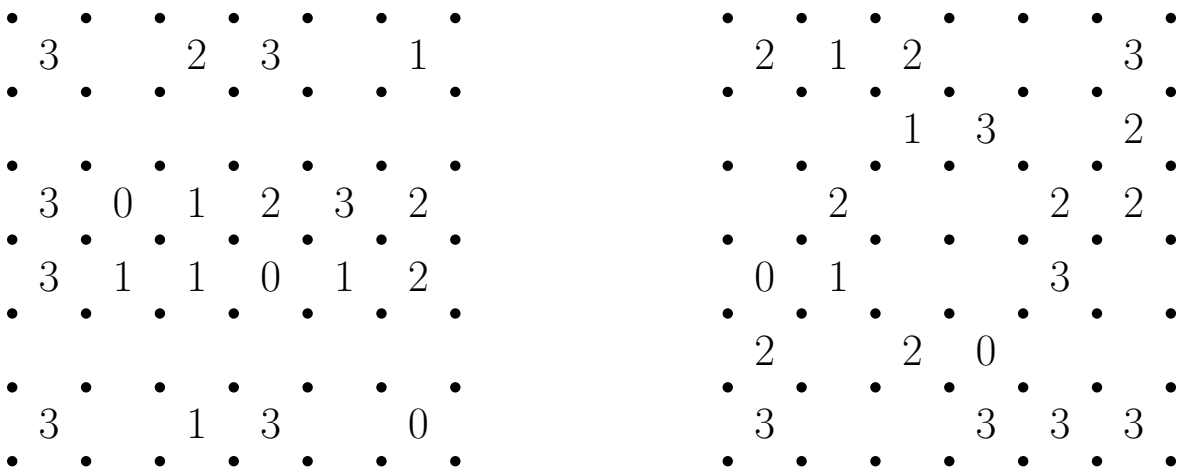


Figure 3.1 – Two *Slitherlink* grids, see rules in Appendix C.2.1.

Part II

Samplers

Chapter 4

A History of Samplers

4.1 Introduction

Sampling refers to the action of randomly drawing solutions from a family of solutions. It is a strong probabilistic approach to estimating quantities. For example, an approach to estimating the number of fishes in a closed lake can be done in two steps [171]: first, N fishes are caught, tagged, and released. A few days later (to allow the fishes to mix), a N fishes are caught again, and the number of fishes marked (say n) is counted. Thus there is an estimated proportion of n/N tagged fishes in the lake, then the estimated total number of fishes in the lake is N^2/n. This sampling approach avoids drying up the lake and killing all the fishes to count them.

In combinatorial problems, the solution space is often too large to be enumerated. In theory, sampling approaches can be used to estimate the number of solutions. Sampling also provides diversity in the solutions returned, for example when diversity cannot be formally defined, or when it is too expensive to compute.

The holy grail of sampling is perfect uniformity, where all solutions have the same probability of being sampled. Weighted sampling is even harder because it allows the users to define their own solution distribution. In constrained problems, very few samplers achieve uniformity. On the other hand, efficient sampling, even if not exactly uniform, can be used as an approximation to generate multiple solutions. This leads to multiple samplers. Figure 4.1 shows a historical timeline of SAT samplers. We can see that sampling constrained problems is an active area of research, as almost half of the samplers in this timeline were created after 2017.

In this chapter, we review the constrained samplers (in SAT and CP) using a thematic and historical approach. In Section 4.2 we present the SAT samplers categorised by the approach they use. In Section 4.3 the CP samplers are presented. Finally, in Section 4.4 we present how to evaluate the randomness of constrained samplers.

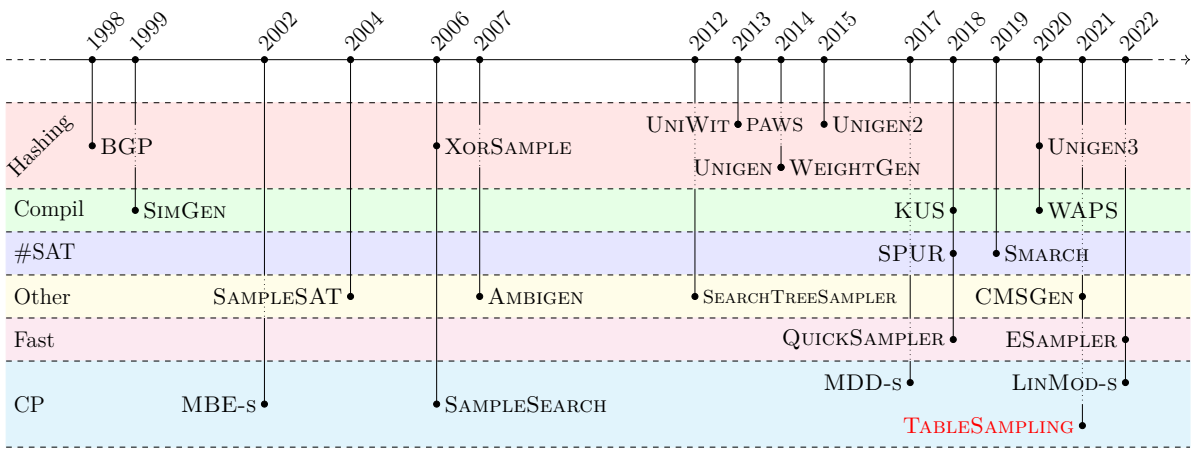


Figure 4.1 – Historical timeline of samplers

1 **Function** *HashingBasedSampler*(F)
  Data: A propositional formula F
  Result: A random assignment of F
2 **repeat**
3 $C \leftarrow$ random hashing constraint
4 $cell \leftarrow F \wedge C$
5 $\mathcal{S} \leftarrow Sols\,(cell)$
6 **until** $0 < |\mathcal{S}| \leq pivot$
7 **return** Random $(\mathcal{S})$

Algorithm 4.1: Outline of a hashing-based sampler

4.2 SAT Samplers

As defined in Chapter 3, Section 3.3, a uniform sampler is defined by a property on the returned solutions rather than by an implementation. This led to broad design ideas. To present these samplers, we grouped them by topic, to emphasise the differences in design, properties, and efficiency.

4.2.1 Hashing Based

Using hashing functions to reduce the solution space is the core idea of many samplers. A simplified outline of such a sampler is given in Algorithm 4.1. The samplers will try to divide the solution space into *small cells*[1] (defined by a *pivot* value in line 6). To create and find such a small cell, a random hashing constraint is generated (line 3), added to

1. Here, "cell" refers to a sub-space of the problem constrained by hashing constraints.

the problem (line 4), and the solutions of that cell are enumerated (line 5). If a small cell is found, one of its solutions is chosen at random. The samplers I present in this section mostly follow this algorithm. They may use different hashing constraints, different *pivot* values, or different stopping conditions. The idea is always to find small cells by using hashing constraints.

Historically, hashing-based samplers originated from work in complexity theory linking the question of approximate counting and uniform generation [81, 191, 92]. Approximate counting allows to know the density of sub-spaces, which helps to know with what probability one should search in those sub-spaces in order to obtain a uniform distribution. This connection is used for example in UNIGEN, which makes calls to APPROXMC, a probably approximately correct model counter (see Definition 18).

In 2006, the $\mathcal{H}_{xor}$ family of XOR constraints was introduced [79] and became the baseline hashing constraints. The years 2013-2015 saw many improvements in the use of these XOR constraints, leading to the current state-of-the-art sampler UNIGEN3.

In this section, the samplers have strong theoretical guarantees of almost-uniformity or near-uniformity.

BGP

One of the first procedures for uniform sampling, BGP [2], is presented in [65]. It uses an r-independent family of hashing functions using $r-1$ degree polynomials over $\mathbb{F}_2$ from $\{0,1\}^n$ to $\{0,1\}^m$, denoted $\mathcal{H}(n,m,r)$. Given a formula F over n variables, and denoting $l = 2\lceil \log_2 n \rceil$, the algorithm searches for a value $i \in \{l, \ldots, n\}$, and a hashing function $h \in \mathcal{H}(n, i-l, r)$ such that $\forall \alpha \in \{0,1\}^{i-l}, |Sols(F \wedge h(x) = \alpha)| < 2n^2$ (i.e. it searches for a hashing function making small cells). A random cell is chosen by picking a random $\alpha \in \{0,1\}^{i-l}$. All the solutions $Sols(F \wedge h(x) = \alpha)$ of this cell are enumerated, an index $j \leq 2n^2$ is chosen, and the j-th solution is returned (if there are fewer than j solutions, the algorithm fails, and should be re-run).

XorSample

XORSAMPLE [79] is the first sampler that uses XOR constraints to sample solutions. On average, XOR constraints divide the number of solutions of the model by two. Then if there are 2^{s^*} solutions to the model, adding s (with s close to s^*) XOR constraints to

2. We keep the name BGP as used in [67] from the initial letters of the authors' names

the model makes it likely to retain a single solution. The authors present two variants. In the first one, if there is more than one solution, new XOR constraints are generated and the process is restarted. In the second variant, all the solutions of the model constrained by the XOR constraints are computed, and one is returned uniformly at random (if there are no solutions, the algorithm fails).

The authors show that when s is close to s^*, the distribution of the solutions is exponentially close to the uniform distribution. However they do not provide any way to estimate this parameter s.

UniWit

The previous two samplers have flaws that make them impractical, either too many calls to SAT solvers have to be done, or some parameters need to be estimated. BGP needs to ensure that *every* cell is small, by counting the 2^{i-l} cells. Also, the threshold for a cell to be considered small is $2n^2$. To run XorSample, an approximation of the number of solutions is required to know how many XOR constraints to add, but the authors do not show how to get such an approximation.

To address these issues, the authors of UniWit [67] noted that only a single cell needs to be checked in BGP, that the limit at which a cell is considered small can be lowered to $2n^{1/k}$, and that the number of XOR constraints added can be increased over time until the cells are small enough. They also use the `CryptoMiniSat` [58] SAT solver, which is optimised for XOR constraints.

Algorithm 4.2 presents the pseudocode of UniWit. It uses a function BoundedSAT(F, n) which returns at most n solutions of F. If there are fewer than *pivot* total solutions, one of them is randomly returned (line 4). Otherwise, the loop (lines 7-11) increases the number of XOR constraints added until a small non-empty cell is found. Then, a random solution is returned (with a chance of failure if the random index j is greater than the number of solutions in the cell).

PAWS

PAWS [75] is a weighted sampler based on an embedding (into a higher dimension problem) and a projection (into small cells). The embedding increases the dimension of the problem $\mathcal{P}$ into a problem $\mathcal{P}'$ so that uniform sampling in $\mathcal{P}'$ is equivalent to weighted sampling in $\mathcal{P}$. The projection step adds XOR constraints to $\mathcal{P}'$ in the hope that the projected problem will have fewer solutions than a selected pivot value. The number of

Function *UniWit*(F, k)
  Data: A propositional formula F on variables $\mathcal{X}$, an integer $k \geq 1$
  Result: A random assignment of F (or a fail $\perp$)
  $pivot \leftarrow \left\lceil 2n^{1/k} \right\rceil$
  $S \leftarrow \text{BoundedSat}(F, pivot + 1)$
  if $|S| \leq pivot$ **then return** $\text{Random}\,(S)$
  $l \leftarrow \left\lfloor \frac{1}{k} \log_2 n \right\rfloor$
  $i \leftarrow l - 1$
  repeat
    $i \leftarrow i + 1$
    $h \leftarrow \text{Random}\,(\mathcal{H}_{xor}\,(n, i - l))$
    $S \leftarrow \text{BoundedSAT}(F \wedge h(\mathcal{X}), pivot + 1)$
  until $1 \leq |S| \leq pivot \vee i = n$
  if $|S| > pivot \vee |S| = 0$ **then return** $\perp$
  else
    $j \leftarrow \text{Random}\,(\{1, \ldots, pivot\})$
    if $j \leq |S|$ **then return** S_j
    else return $\perp$

Algorithm 4.2: UniWit: sampling using XOR constraints

XOR constraints to add is chosen by a procedure making an accurate guess with high probability.

WeightGen

WeightGen [70] is a weighted sampler using recent advances made in approximate model counting [68]. It proposes a weighted model counting algorithm WeightMC adapted from ApproxMC [68]. A call to WeightMC gives an approximation of the sum of the weights, which is used to approximate a number q of XOR constraints to add. Then i constraints are added where i varies from $q - 3$ to q. As XOR constraints are added, the sum of the weights of the remaining solutions is computed, and if it is between a low and a high threshold, one random (weighted) solution is returned.

Unigen

Unigen [82] is a family sampler that has seen many improvements over the years.

Unigen-1 [69] Released at the same time as WeightGen, Unigen-1 shares the same ideas, without the weights. It uses ApproxMC to compute an approximate model count, and uses this count to know a value q of how many XOR constraints to add. Varying the number of constraints added from $q-3$ to q, if the number of solutions remaining in the constrained model is between a low and a high threshold, a random (and uniform) solution is returned.

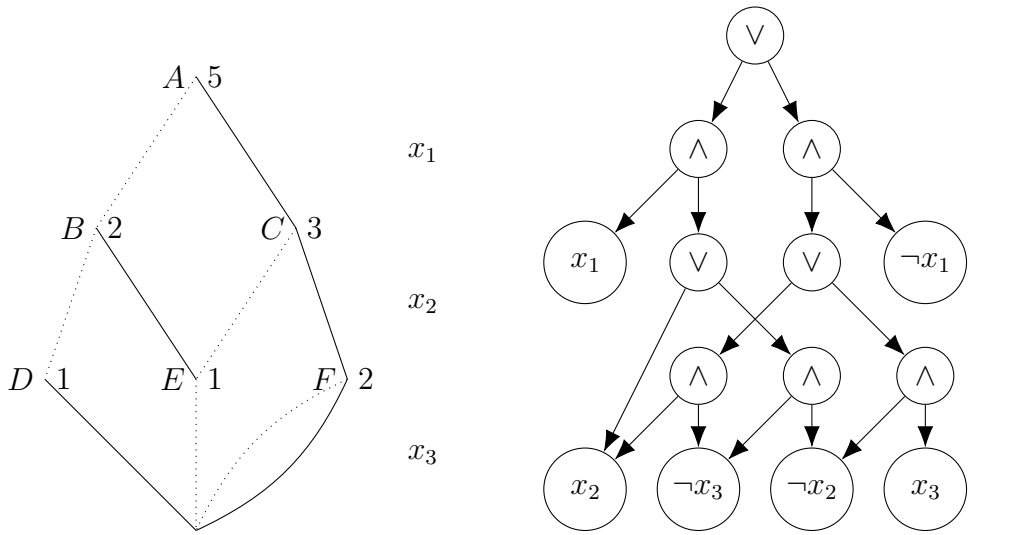
Unigen-2 [71] Unigen-2 focuses on running time optimisation while maintaining the theoretical guarantees of Unigen-1. An algorithm is designed to improve the computation of the parameters (instead of calling ApproxMC). Parallelization is considered to improve running time (and because previous samplers are not easily parallelizable). Sample generation in Unigen-1 picked one solution from a set of at least $loThresh$ solutions ($loThresh$ is the low threshold defining acceptable small cells). Instead, Unigen-2 returns all the solutions. Then, if N solutions are sought on k threads, performing $\frac{N}{k \cdot loThresh}$ calls to the sampler on each thread (in parallel) generates N solutions.

Unigen-3 [89] The efficiency of a SAT solver directly impacts the efficiency of the associated SAT sampler. The SAT solver `CryptoMiniSat` [58] is dedicated to CNF formulas to which XOR constraints have been added. In [89], the authors present improvements to the SAT solver to better handle with the XOR constraints, called `BIRD2` (for Blast, In-process, Recover, Detach, and Destroy, the five steps of the integration of the XOR constraints in the SAT solver).

For model counting and sampling, an improvement is presented using the previously found solutions. This is due to the improved generation of XOR constraints in [82] for Unigen-2. When increasing the number of constraints, instead of re-generating every constraints, it is possible to re-use the previous XOR constraints and generate a single new one. This means that the new solutions will be a subset of the previous solutions. On average, if there were $thresh$ solutions in the previous round, $thresh/2$ solutions will satisfy the new XOR constraint. Starting by checking if the solutions satisfy the new constraint avoids making unnecessary calls to the SAT solver.

4.2.2 Compilation-Based

The class of instances on which an algorithm works has a strong influence on its complexity. For example, SAT solvers have remarkable performance on industrial instances [5].



(a) BDD representation (b) d-DNNF representation

Figure 4.2 – Two representations of the formula $(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$

There are also classes of instances, such as 2-SAT, where the problem is polynomial.

Compilation-based techniques work in two steps. First, the compilation translates the problem (often in CNF form) into a new structure. Then, an efficient algorithm is applied to this structure to find a solution to the initial problem.

The main disadvantage of this technique is that the compilation phase is very expensive and can produce a structure that is exponentially larger than the original formulation. However, the structure only needs to be generated once, and all the calls to solve the problem can then be performed efficiently. After the compilation phase, samples can be generated with the strong guarantee of exact uniformity.

Examples of the two representations used by these samplers are shown in Figure 4.2. The formula used has 5 solutions $(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 0)$, and $(1, 1, 1)$.

SimGen

SimGen [95] uses a *Binary Decision Diagram* (BDD) to represent the set of solutions of a problem. An example of a BDD is shown in Figure 4.2a. A BDD is a layered directed acyclic graph (DAG) where each layer represents a variable. A path from the top to the bottom is a solution where if a dotted edge is taken, the variable is instantiated to 0, and if a plain edge is taken, the variable is instantiated to 1.

With a bottom-up traversal, the number of paths from a node to the bottom of the

BDD can be computed, representing the number of partial solutions from that node. This information can then be used in a top-down traversal to know with what probability to take each edge, leading to a uniform distribution.

Example. *In Figure 4.2a we show a BDD where the nodes have been named (A to F). A path from the top to the bottom represents a solution of the formula. For example, the path through A, B, and E represents the solution* $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 0\}$. *The values next to the nodes are the number of extensions of the current partial assignment from this node to the bottom. For example, C (representing the partial assignment* $x_1 \mapsto 1$*) can be extended to 3 solutions. The number of assignments for every node can be computed in a bottom-up pass.*

These counts can then be used to know with what probability to choose each value for the variables. For example, $x_1 \mapsto 1$ *can be extended to three solutions, and* $x_1 \mapsto 0$ *can be extended to two solutions. When sampling we choose the value 1 for* x_1 *with probability* $2/5$. *If the value 1 is chosen, then* $x_2 \mapsto 0$ *can be extended to one solution, and* $x_2 \mapsto 1$ *can be extended to two solutions. We choose the value 1 with probability* $2/3$.

KUS

KUS [88] uses more modern compilation techniques to compute a d-DNNF representation. Such a representation is a DAG with `or` and `and` nodes, and literal leaves. An example of a d-DNNF representation is given in Figure 4.2b. In addition, d-DNNFs are *deterministic*, i.e. the operands of `or` nodes are mutually inconsistent, and *decomposable*, i.e. the operands of `and` nodes should be expressed in a mutually disjoint set of variables. The compilation tool D4 [176] is used to obtain the d-DNNF representation of the input formula.

After the compilation a bottom-up annotation phase is performed to compute for each node the number of solutions and the set of variables in the sub-formula. Then, samples can be generated using a top-down pass on the annotated d-DNNF.

Example. *Figure 4.2b shows the representation of the example formula using a d-DNNF. In a d-DNNF, leaf nodes are literals, inner nodes are operations (*$\vee$ *or* $\wedge$*) and their children are the operands. The exact transcription of the example d-DNNF is the formula*

$$\Big(x_1 \wedge \big(x_2 \vee (\neg x_2 \wedge \neg x_3)\big)\Big) \vee \Big(\neg x_1 \wedge \big((x_2 \wedge \neg x_3) \vee (\neg x_2 \wedge x_3)\big)\Big) ,$$

which is equivalent to the initial formula.

WAPS

WAPS [80] is an extension of KUS with three improvements : weighted, projected, and conditioned sampling.

- WAPS allows weighted sampling. The weights are defined as a multiplicative literal-weight function, i.e. the weight of an instantiation is the product of the weights defined over the literals. The annotation phase uses this weight function to compute the weights of the sub-formula represented by the node, and the sampling can be done in the same way as in KUS.
- It allows projected sampling, i.e. given a formula G on variables $\mathcal{X}$ and $\mathcal{Y}$, the projection on the set $\mathcal{X}$ is defined as $F(\mathcal{X}) = \exists \mathcal{Y}, G(\mathcal{X}, \mathcal{Y})$. This is useful, for example, when the formula has been rewritten and there is not a one-to-one solution correspondence between the initial and the new formula. This projection is enforced using the d-DNNF compiler Dsharp [179].
- It allows conditioned sampling, i.e. sampling with some literals fixed. This is done by modifying the weight function to give a weight of 0 to solutions where the literals do not have the chosen value. Going through the annotation phase again allows to generate conditioned samples.

4.2.3 #SAT-Based

The two samplers presented in this section are based on the #SAT problem, i.e. counting the number of satisfying assignments of a SAT problem. They could arguably be classified as compilation-based techniques, since they implicitly traverse a tree-like structure. As such, they are exact uniform samplers.

SPUR

SPUR [64] is an adaptation of the #SAT solver sharpSAT [91] to allow for uniform sampling. We will first present sharpSAT and then show the modifications made in SPUR.

sharpSAT sharpSAT is based on the DPLL algorithm [14], and on #SAT ideas such as *component decomposition* and *component caching*. sharpSAT improves the component caching by encoding the components differently, greatly reducing the size of the encoding. It also uses an algorithm for finding failed literals that is more suited to #SAT.

```
1  Function sharpSAT(F)
       Data: A propositional formula F
       Result: The number of satisfying assignments of F
2      BCP(F)
3      if IsCached(F) then return CachedCount(F)        // Cache-hit leaf
4      if Clauses(F) = ∅ then return 2^|Var(F)|
5      if Unsat(F) then return 0
6      C_1, ..., C_k ← ComponentDecomposition(F)
7      if k > 1 then                                    // Component decomposition
8          for i = 1 to k do
9              Z_i ← sharpSAT(C_i)
10         Z ← ∏_{i=1}^{k} Z_i
11     else
12         v ← BranchVariable(F)
13         Z ← sharpSAT(F ∧ v) + sharpSAT(F ∧ ¬v)
14     AddToCache(F, Z)
15     return Z
```

Algorithm 4.3: sharpSAT: model counting with component decomposition and caching

The outline of sharpSAT is presented in Algorithm 4.3. Basically, it is a recursive algorithm that performs boolean unit propagation (line 2), splits the formula into sub-formulas and solves them recursively. If a sub-formula has no unsatisfied constraints, there may be uninstantiated variables remaining. The number of satisfying assignments is then $2^{nbFreeVars}$, where $nbFreeVars$ is the number of uninstantiated variables. If there is an empty clause, then there are no solutions to the sub-formula.

There are two ways a formula can be split. After some variables have been instantiated, the constraint network may be disconnected. The connected components can be counted recursively. The count of the main formula is then the product of the counts of the formulas for each component (line 10). Otherwise (if the constraint network is connected), a variable is selected and branched on. The count of the main formula is the sum of the counts of the two sub-formulas (line 13). The branching strategy that chooses the variable is the VSADS heuristic from [53] as it is tailored for model counting.

In addition to component decomposition, sharpSAT also implements a cache storage of the number of assignments of the computed sub-formulas. When a count is computed, it is added to the cache (line 14). When a formula to be computed is already present in the cache, the count is returned directly (line 3). This is implemented using an efficient

representation of formulas. A forgetting process is also implemented to prevent the cache size from exceeding a user-defined threshold. A score is given to the formulas in the cache each time they are hit, and all the scores are divided periodically.

SPUR SPUR is built on top of sharpSAT and has the same structure. We show how to modify Algorithm 4.3 to sample uniformly. We first consider a simplified version of sharpSAT without caching. Instead of simply returning the number of solutions of the formula, SPUR also returns a uniformly selected partial assignment. These assignments can come from three places:

- Line 4: when there are no more constraints, a random value is chosen for all the uninstantiated variables.
- Line 10: different partial assignments from the component decomposition can be merged together.
- Line 13: when a branching, one of the branches has to be selected with a probability proportional to its count.

Component caching actually does not actually complicate this process. The authors noted that there is no issue of non-independence when caching the samples because a cached sample can only be used once in the solution. Caching can then be added without further consideration.

Sampling multiple solutions (say k) at once adds a level of difficulty. The authors used a technique called *reservoir sampling* to deal with multiple solutions. The idea is to store (in the cache) for each sub-formula a "reservoir" of k partial assignments. Then, when merging partial assignments (line 10) or choosing a sub-branch (line 13), the recursive partial assignments can be selected from the reservoirs. For the exact procedure, we refer the reader to the original article [64] where the authors show how to compute the number of solutions to sample from each reservoir to ensure uniformity of the sampling.

Smarch

Smarch [84] is a uniform sampler based on #SAT, but it differs from SPUR in several ways. The algorithm is presented in Algorithm 4.4. Instead of sampling as the counting tree is traversed, calls are made to a #SAT solver (sharpSAT). When there are no more constraints, a random assignment of the remaining variables can be returned. Otherwise, the *cube-and-conquer* strategy [26] (line 9) is used to split the formula into sub-formulas that are counted, and one is chosen to sample from.

```
1  Function SMARCH(F, n)
       Data: A propositional formula F, an integer n
       Result: n distinct uniformly sampled assignments of F
2      count ← |Sols(F)|
3      rSet ← n distinct random integers from [1, count]
4      return {SMARCHSAMPLE(F, r, ∅) | r ∈ rSet}
5  Function SMARCHSAMPLE(F, r, s)
       Data: A propositional formula F, an integer r, and a partial assignment s
       Result: The r-th satisfying assignment of F ∧ s
6      if BCP(F ∧ s) has no constraint then
7          return s∧ r-th assignment of the uninstantiated variables
8      else
9          cubes ← CUBEDECOMPOSITION(F ∧ s)
10         for cube ∈ cubes do
11             cs ← |Sols(F ∧ s ∧ cube)|
12             if r ≤ cs then
13                 return SMARCHSAMPLE(F, r, s ∧ s)
14             else
15                 r ← r − cs
```

Algorithm 4.4: SMARCH: sampling with cubes decomposition and counting

Compared to SPUR, no caching is done, but the samples are guaranteed to be different (line 3).

4.2.4 Other Samplers

In this section, we present solvers that do not fit into any of the other categories. The three samplers presented here are designed to return solutions from a distribution close to the uniform distribution, but no theoretical proofs are given (and evaluations [66, 74, 90] show that they are not exact uniform samplers).

SampleSAT

SAMPLESAT [93] is based on the WALKSAT solver [87], which is classified as a Monte-Carlo Markov Chains (MCMC) approach. MCMC methods are among the best known methods for sampling combinatorial spaces and have many variations. For a general presentation of MCMC methods we refer the reader to [185].

In the case of propositional formulas, a random walk starts from a random instantiation

(which does not satisfy the constraints) and flips the values of variables until all the constraints are satisfied. The flip is chosen by picking a literal of an unsatisfied clause. On 2-SAT, in [180], the authors showed that such a random walk finds a solution with high probability in $\mathcal{O}(n^2)$ steps (where n the number of variables). On 3-SAT a similar result is given in [188] but the number of steps is exponential: $\mathcal{O}(1.334^n)$, still better than the whole space enumeration in $\mathcal{O}(2^n)$. In WalkSat, the algorithm picks a variable minimising the number of unsatisfied clauses. Ties are broken randomly.

In SampleSAT the authors analyse the behaviour of WalkSat on the sampling distribution. They find that it is highly non-uniform. They propose to interleave this algorithm with *simulated annealing* steps. The procedure chooses a random variable to flip, and computes ΔC the increase in the number of satisfied clauses. The algorithm then performs the flip with probability $e^{-\Delta C/T}$ (where T is the temperature parameter, tuned by the authors and set to 0.1). This means that SampleSAT allows to flip variables that increase the number of unsatisfied clauses. This simulated annealing step greatly improves the distribution of the samples.

Ambigen

Ambigen [174] is a sampler for mixed Boolean/integer formulas, where the integer variables can only be constrained by linear inequalities. It is based on Monte-Carlo Markov Chain methods (similar to SampleSAT) and combines Metropolis-Hastings, Gibbs and WalkSat moves to randomly generate solutions.

It is based on a random move procedure that finds a new solution from an initial one. The initial solution is modified by flipping a boolean variable, or changing the value of an integer variable. If this new assignment does not satisfy the constraints, Metropolis and WalkSat moves are performed until all the constraints are satisfied. This move procedure is then used to generate new solutions. Performing multiple moves changes the solution enough so that the new solution has little correlation with the initial solution.

SearchTreeSampler

SearchTreeSampler [74] uses a SAT solver as a black box. The principle is to extend a set of *pseudo-solutions* until all the variables have been instantiated.

Definition 23 (Pseudo-solution)**.** A *pseudo-solution* of level m is an assignment to the first m variables that can be completed to form a solution.

1 **Function** SearchTreeSampler(F, k, l)
  Data: A propositional formula F with n variables, k a number of solutions to sample, l step size for each iteration
  Result: Between k and $2^l k$ solutions from F (if enough exist)
2 **if** *F is not satisfiable* **then return** $\emptyset$
3 $\Phi_0 \leftarrow \{\emptyset\}$ `// empty assignment`
4 $L \leftarrow \lceil \frac{n}{l} \rceil$
5 **for** $i = 1, \ldots, L$ **do**
6 $\Phi_i \leftarrow \emptyset$
7 **for** $j = 1, \ldots, \min(k, |\Phi_{i-1}|)$ **do**
8 $s_j \leftarrow$ sample from Φ_{i-1} without replacement
9 $\Phi_{s_j} \leftarrow$ pseudo-solutions of level $i * l$ with s_j as ancestor `// Calls the SAT solver`
10 $\Phi_i \leftarrow \Phi_i \cup \Phi_{s_j}$
11 **return** Φ_L

Algorithm 4.5: SearchTreeSampler: layered sampling

Remark. *The* first *variables do not have to be the first ones defined in the problem. An order can be specified but for the sake of simplicity (and without loss of generality) we present the algorithm without specifying an order.*

The algorithm is presented in Algorithm 4.5. A set of pseudo-solutions Φ is grown iteratively from the singleton of the empty (level 0) pseudo-solution. At each step at most k pseudo-solutions are sampled from this set, and each of these solutions is extended to all the possible pseudo-solutions at a level increased by a parameter l. This step, line 9, can be solved by iteratively solving the problem $F \wedge s_j \wedge \bigwedge_{s \in \mathcal{S}} \neg s$ where s_j is the current pseudo-solution, and $\mathcal{S}$ is the set of pseudo-solutions being generated. The algorithm returns at least the required k solutions.

There is no actual guarantee on the uniformity at the end of the search, but there is an iterative property.

Property 5. *Suppose that Φ_i is a uniform sample of pseudo-solutions of level i. Let s, s' be two solutions of Φ_{i+1}, then*

$$\frac{1}{1+\epsilon} \leq \frac{\mathbb{P}(s)}{\mathbb{P}(s')} \leq 1 + \epsilon$$

with $\epsilon = \frac{2^l - 1}{k}$, depending on the parameters k and l.

This property states that if the solutions in Φ_i are uniformly distributed in the set of pseudo-solutions (of level $i \times l$), then the solutions in Φ_{i+1} are *almost*-uniformly distributed. Then the solutions in Φ_{i+2} are *almost-almost*-uniformly distributed, and so on. At the end of the process, after $\lceil \frac{n}{l} \rceil$ iterations, the distribution may have drifted too far from the uniform, and may be skewed.

CMSGen

CMSGen [78] is a sampler based on the conflict-driven clause-learning [57] (CDCL) algorithm of modern SAT solvers (using `CryptoMiniSat`). Its design is very simple: it uses RandomSearch with clause learning and restarts. It picks a random unassigned variable and assigns it a random value. Unit propagation is performed, and if all variables are assigned the assignment is returned. If a conflict is found the solver's conflict analysis learns new clauses representing the conflict, and performs a back-jump to continue the search. If too many conflicts are learned, there is a deletion mechanism to reduce the formula. CMSGen also performs periodic restarts to allow the search to escape from some parts of the search space.

It is quite easy to show the non-uniformity of CMSGen, since it is based on RandomSearch (as we proved in Chapter 2 Section 2.4.2), and as shown in an issue in CMSGen's repository. [3] Despite this non-uniformity, Barbarik accepts CMSGen as a uniform sampler. This led to the design of ScalBarbarik which rejects CMSGen (see section 4.4 about evaluation of SAT samplers).

4.2.5 Efficiency Oriented

The samplers presented so far often provide strong guarantees at the cost of a long running time. In some cases, these strong guarantees are not needed, and a lot of random (well distributed but not necessarily uniform) solutions are required in a short time. This has led to the design of efficient samplers, that generate solutions non-uniformly, but quickly.

QuickSampler

QuickSampler [73] is a sampler, based on *atomic mutations*. Given two solutions σ_a and σ_b, an atomic mutation is $\delta = \sigma_a \oplus \sigma_b$. Given a third solution σ_c, QuickSampler's

3. `https://github.com/meelgroup/cmsgen/issues/3`

idea is to look at the assignment $\sigma_c \oplus \delta$ that *may* be a solution.

QuickSampler starts with an initial solution σ. Starting from this solution, it iterates over all variables x_i, $1 \leq i \leq n$. For each variable, it finds an instantiation σ_i, that differs from σ on the variable x_i ($\sigma(x_i) \neq \sigma_i(x_i)$), using MaxSAT queries. Each of these instantiations give an atomic mutation δ_v, and the set of all atomic mutations found this way is denoted Δ^1_σ. From this set Δ^1_σ, atomic mutations can be combined to produce new mutations of higher order, giving the sets Δ^k_σ. Then the sets $\Sigma^k_\sigma = \{\sigma \oplus \delta \mid \delta \in \Delta^k_\sigma\}$ are generated, corresponding to all the assignments mutated from the initial σ using mutations. Experimentally the authors show that, with high probability, the assignments in Σ^k_σ satisfy the constraints, leading to new solutions.

To generate the assignments σ_i, the MaxSAT queries are done by adding soft constraints on the values of σ, and forcing v to be different. To select σ at the beginning, a random assignment is chosen and its closest solution is found using the same MaxSAT query.

Some tweaks can be made to improve the algorithm and its properties. Δ^k_σ can be computed as soon as one new δ is found. The sampler can also be easily modified to sample projected formulas.

ESampler

ESampler [94] is a method for efficiently generating solutions from an oracle sampler. It is based on the same general idea as QuickSampler, but it will never produce invalid assignments.

Algorithm 4.6 presents ESampler, and its the derivation function. The main function generates an assignment from an oracle sampler (the authors use QuickSampler and Unigen-3). From this assignment, a set of solutions is derived, and added to the generated set.

The derivation procedure takes as input the formula, an assignment, and a maximum number of solutions to generate. A queue Q is created (line 11), initially containing only the input assignment. Iteratively, an assignment is popped from this queue (line 13) and used to generate new assignments. These assignments are created by flipping the value of one variable (line 15). If a new assignment satisfies the constraints (line 16), it is added to the set of derived solutions, and to the queue. When enough assignments have been generated (less than a maximum number N_{max}), or when the queue is empty (all the sub-space has been explored), the derived set is returned.

```
1  Function ESampler(F, k, Sampler, N_max)
       Data: A propositional formula F, a desired number of solutions k, a sampler
             Sampler, and a maximum derivation size N_max.
       Result: At least k distinct assignments of F.
2      S ← ∅
3      repeat
4          σ ← Sampler(F)
5          S_derived ← Derive(F, σ, N_max)
6          S ← S ∪ S_derived
7      until |S| ≥ k
8      return S
9  Function Derive(F, σ_init, N_max)
       Data: A propositional formula F, an initial assignment σ_init, and the
             maximum number of assignments to generate N_max.
       Result: At most N_max distinct assignments of F.
10     S_derived ← {σ_init}
11     Q ← {σ_init}
12     while |Q| ≠ 0 ∧ |S_derived| < N_max do
13         σ ← Q.pop()
14         for 1 ≤ i ≤ n do
15             σ_i ← σ[x_i ↦ ¬σ(x_i)]
16             if σ_i ∉ S_derived and σ_i satisfies F then
17                 S_derived ← S_derived ∪ {σ_i}
18                 Q.add(σ_i)
19     return S_derived
```

Algorithm 4.6: ESampler: derived samples

ESampler offers a compromise between uniformity (only calling the sampler and not the derivation procedure with $N_{max} = 1$) or efficiency (a high value of N_{max} to derive many assignments). When using the derivation procedure, ESampler is able to generate solutions orders of magnitude faster than other samplers.

4.3 CP Samplers

The samplers presented so far are designed to sample SAT problems. Extending these samplers to CP is not an easy task because CP deals with larger domains and broader a

language of constraints (such as global constraints). We present here CP samplers [4].

4.3.1 MBE-s

MBE-s [72] is to our knowledge the first sampler working on constraint satisfaction problems. It transforms the constraint network into a belief network, a structure representing conditional probability tables, i.e. $\mathbb{P}(X = a \mid X_{j_1} = a_1, \ldots, X_{j_k} = a_k)$. To do so, a framework called *bucket-elimination* is used, with a time and space exponential algorithm. If n is the number of variables, then n buckets are created and the i-th bucket stores all the constraints whose last variable (in a chosen order) is X_i. The buckets are then used to generate functions computing conditional counts, i.e. number of solutions extending a partial assignment. These conditional counts can then be used to compute the conditional probability tables. Samples can be generated by instantiating the variables in the given order, using the conditional probability to weight the different values of the variable.

This bucket elimination is time and space exponential unless the induced width of the constraint graph is bounded. To reduce the time and space requirements, an approximation scheme called *Mini-Bucket Elimination* (MBE) is used. Instead of computing the exact counts, MBE computes an upper bound of the conditional counts. A bounding parameter can be chosen to control the tightness of the bound, trading efficiency for uniformity.

4.3.2 SampleSearch

SampleSearch [76] uses the same idea of computing the conditional probability tables. Instead of using the Mini-Bucket Elimination, the author uses IJGP [168], a different belief propagation algorithm. Instead of working on buckets, IJGP works on a structure called the join-graph, and iteratively propagates information on it to compute the conditional probability tables (hence the name, Iterative Join-Graph Propagation). IJGP is also exponential in time and space, and an approximation scheme is proposed, IJGP(i) by limiting the number of variables during the computations.

The authors note that the approximation scheme removes guarantees that the samples will be solutions. To ensure that all samples generated are solutions to the problem, the sample generation is modified. If a sample is not a solution, a back-jump is performed,

4. When the samplers presented here have not been given a name by the authors, we name them after the technique they use to sample.

and the (approximate) probability distribution is updated. A no-good is also added to prevent the algorithm from searching this space again.

SampleSearch was later improved in [77] in two ways. First by using a Metropolis-Hasting method, which performs a random walk on the set of samples. It starts from an initial sample, and changes to a new one with a given probability (depending on the distributions of the samples). A second method is proposed using Sampling/Importance Resampling. The idea is to first generate N samples, and then extract (re-sample) the M desired solutions using a weighted distribution. The resulting distribution converges to the uniform distribution as N grows.

4.3.3 MDD-s

MDD-s [85] is a compilation-based technique adapted to CP. Multi-valued Decision Diagrams (MDDs) are an extension of BDDs as used in SimGen [95] already presented in section 4.2.2. As for the BDDs, an annotation phase is performed to compute the weights of the arcs, and then sampling can be performed in a top-down linear pass on the MDD.

4.3.4 Recent samplers: TableSampling and LinMod-s

In 2021, I proposed TableSampling [1], a sampling algorithm based on hashing techniques (presented in section 4.2.1) adapted to constraint programming. I also showed experimentally that using linear modular equality constraints (instead of `table` constraints) makes the sampler uniform. Simultaneously, linear modular equalities (and inequalities) were used to design a sampler, LinMod-s [86].

I present TableSampling in detail in the following Chapter 5. I also postpone the presentation of LinMod-s to the same chapter, in Section 5.8, in order to have a precise comparison of the two approaches.

4.4 Evaluation

As with any algorithm, samplers should be tested on different instances, and their behaviour should be studied. As they are random by nature, multiple runs should be done to get an average (or median) value, for example for the running time.

Testing the quality of the randomness is more difficult. By definition, any event, even if unlikely, can occur. This includes events where the sampler does not seem uniform (for

example, returning the same solution multiple times).

4.4.1 Examples of Evaluations

To assess the quality of the randomness, the authors evaluate their samplers using various methods. Of the 24 samplers presented, a distribution test was performed on 14 of them. These tests are often performed on smaller problems on which the whole solution set can be enumerated, so that the sampler can return the same solution several times in order to know precisely the distribution of solutions.

Solution Distribution

One of the first ways of evaluating the randomness is simply to print out the frequency of each solution (i.e. the distribution). If the sampler is far from uniform, this test allows to see which spaces are sampled more than others (such as done in Figure 2 of [93]).

Solution Counts

If the distribution of solutions seems uniform, one can look at the distribution of the number of occurrences of each solution. Formally, let us consider a uniform sampler over the set $[1, N]$ and draw from it M times (through random variables $X_k, 1 \leq k \leq M$). We count the number of occurrences $Occ_i = |\{X_k | X_k = i, 1 \leq k \leq M\}|$. This number of occurrences should follow a binomial distribution with parameter M and $p = 1/N$. If the experimental distribution is far from the binomial distribution, then the sampler may not be uniform.

To have a quantitative value of the difference between two distributions, the Kullback-Leibler (KL) divergence can be used.

Definition 24. Given two probability distributions P and Q over a set $\mathcal{S}$, the *Kullback-Leibler* (KL) divergence is defined as

$$D_{KL}(P \mid Q) = \sum_{s \in \mathcal{S}} P(s) \log \left(\frac{P(s)}{Q(s)} \right) .$$

This divergence is equal to 0 when the distributions are the same, and increases when the distributions are different. This KL divergence can also be used on the distribution of the solutions to compare directly to the uniform distribution.

χ^2 Test

More powerful statistical tests can also be used. Pearson's χ^2 test [181] is a test that can state with high confidence that a sampler does not follow a given distribution. This test is applied to TableSampling and is presented in detail in the following Chapter 5. This test takes advantage of the number of samples drawn to give a more confident answer.

4.4.2 Evaluation Tool

The evaluation approaches presented previously need to be able to sample all the solutions multiple times. In [164], the authors show that if one only has access to samples, then at least $\Omega\alpha\sqrt{|Sols(\mathcal{P})|}$ samples are required to give an experimental guarantee (with α a value depending on the strength of the guarantees). This requirement is completely impractical in some instances (for example, many instances of feature models we used in Chapter 7 have more than 10^{120} solutions).

More recently, an evaluation tool, Barbarik [66], has been designed specifically for SAT samplers (and improved in two subsequent versions). It overcomes this *testing curse of dimensionality* by designing formulas with two solutions (or two classes of solutions). We present it in this section.

Barbarik

In order to prove uniformity, the samplers must sample many solutions. However, if the instance has few solutions, then this task becomes easy. The extreme case is when the instance has only 2 solutions (or 2 sets with the same number of solutions).

Algorithm 4.7 presents a simplified version of Barbarik to outline how it works. We refer the reader to the original article [66]for the complete presentation, including full pseudocode, implementation details, and proofs. Barbarik generates a sample σ_1 using the sampler under test, and another sample σ_2 using a uniform sampler (such as SPUR). Then, it creates a new formula using these two samples such that this formula has 2τ solutions, where τ solutions correspond to σ_1, and the other τ solutions correspond to σ_2 (by projecting over a subset of variables, simplified here). It then generates samples using the sampler under test on the newly constructed formula. It counts the number of samples generated that correspond to σ_1, and if this number is too far from 1/2 it rejects the sampler. In the full algorithm, this process is repeated several times and if the sampler passes all the tests, then Barbarik returns $ACCEPT$.

```
1  Function BarbarikSimple(A, U, φ)
       Data: A sampler A under test, a uniform sampler U, a formula φ
       Result: ACCEPT (resp. REJECT) the uniform (resp. non-uniform) sampler
2      σ1 ← A(φ)
3      σ2 ← U(φ)                                   // different from σ1
4      φ̂ ← Kernel(φ, σ1, σ2)
5      S ← N samples from A(φ̂)
6      b ← |{σ ∈ S | σ = σ1}|
7      if b < (1−C)/2 ∨ b > (1+C)/2 then
8          return REJECT
9      else
10         return ACCEPT
```

Algorithm 4.7: Barbarik (simplified): sampler evaluation

Barbarik2

Barbarik2 [83] improves Barbarik by allowing the testing of weighted samplers. It is based on the same idea, except that when sampling from the new formula ($\widehat{\phi}$, with the two classes of solutions), the sampling is weighted, so the rejection criterion depends on the weights of σ_1 and σ_2.

ScalBarbarik

ScalBarbarik [90] was designed after CMSGen which was shown to be accepted by Barbarik but can easily be proven to be non-uniform. Compared to Barbarik, the generation of the new formula is modified in ScalBarbarik. The authors noted that in CMSGen, when the solver spends too much time in a sub-space, restarts are performed. This means that solutions in hard sub-spaces are less likely to be selected. ScalBarbarik generates a formula containing two sub-spaces : one corresponds to the first solution σ_1 and is "easy", i.e. solutions are found quickly, and the other one is "hard", i.e. solutions require more computations to be found, leading to possible restarts. A hardness parameter can be tuned to have an easier or harder formula. ScalBarbarik is shown to reject CMSGen while still accepting Unigen-3 as a uniform sampler.

Chapter 5

Table Sampling

This chapter is taken from my work on TableSampling, which appears in two publications: first in the CP 2021 conference [1], and an extended version in the Constraints journal [2]. A humorous trailer of the conference presentation can be found online [a], as well as the full presentation [b]. The same work was also accepted and presented in the CP 2021 doctoral program, and in the JFPC 2021 conference, where I received the young researcher award [c]. Since the publication, TableSampling has been added to `choco-solver` (version 4.10.9).

a. `https://www.youtube.com/watch?v=Ss4A6OaG_sg`
b. `https://www.youtube.com/watch?v=iX0d_7E-oIc`
c. `https://www.i3s.unice.fr/jfpc_2021/prix/`

5.1 Introduction

Using constraint satisfaction as a core technique, constraint solvers have been enriched with various additional properties, such as optimisation (even with multiple objectives [152]), user preferences [51], diverse solutions [24], robust solutions [23], etc. However, there are very few works about solution randomisation in CP solvers.

In the previous chapter, we introduced eighteen SAT samplers, but only three CP samplers. Moreover, these CP samplers are not designed as improvements to CP solvers, but rather as separate algorithms: MBE-s [72] and SampleSearch [76] transform the constraints into a belief network, and MDD-s [85] transforms the constraints into a MDD. These three samplers do not benefit from improvements in CP solvers (such as a better running time, or new constraints).

In this chapter, we propose a method for sampling solutions to a constraint problem, without modifying its model, and using a CP solver as a black box. This work is motivated by many situations where the user of a constraint solver needs randomised solutions: to facilitate user feedback and decision making (by providing a variety of solutions, represen-

tative of the solution space), to ensure fairness (to avoid patterns in consecutive solutions, for instance in planning problems), or to provide solution coverage (for instance in test generation problems).

Currently, a straightforward way to randomly sample solutions with a CP solver is to use RandomSearch, i.e. to randomly select a variable and a value as an enumeration strategy. However, this strategy does not return uniformly drawn solutions (uniformly within the solution set). Another major drawback of this technique is that RandomSearch replaces the strategy that may have been chosen or built for the problem, and this is likely to increase the solving time.

Our approach is inspired by Unigen [82], an approximately uniform sampling algorithm for SAT, adapted to the CP framework. The idea is to split the search space by adding random hashing constraints, until only a small, tractable number of solutions remain. There is no need to replace the strategy and the sampling can be done among the remaining solutions. Our algorithm also features a dichotomic variation.

The chosen family of random hashing constraints has a strong impact on the running time. To keep it reasonable, we choose to randomly generate table constraints [15], which are implemented in all constraint solvers. We rely on their extensional representation of valid tuples to generate, at low cost, a multivariate uniform distribution.

We implemented our proposal on top of `choco-solver` [46] and compare it to RandomSearch on a broad benchmark, built from the annual MiniZinc competition. We show that our approach using the table constraints improves, in practice, the quality of the randomness compared to RandomSearch, while also sampling more problems.

We also apply our algorithm with linear modular equalities [41], which are hashing constraints with stronger theoretical properties in terms of randomness, but harder to propagate. On our benchmark set, using linear modular equalities gives a better randomness quality compared to the table constraints, as it provides a uniform sampling. The disadvantage is a longer running time.

Outline In this chapter, we first recall the important related works and the definitions in Section 5.2. We then introduce our new sampling approach, TableSampling, in Section 5.3. In Section 5.4 we discuss some design choices, and some properties of TableSampling. The experiments are divided into three sections: first we present the methodology in Section 5.5, then we perform preliminary experiments in Section 5.6, and we finally show experiments on a benchmark of instances of the MiniZinc challenge in

Section 5.7. In Section 5.8 we present the LinMod-s sampler (which was not presented in the previous chapter), and highlight the differences with our approach.

5.2 Background

In this section, we recall the related works and the background needed for this chapter. Most of the related works consist of samplers, which were already presented in the previous Chapter 4, and the background was already introduced in Chapters 2 and 3.

5.2.1 Related Works

The question of sampling combinatorial problems is central in hardware/software verification and testing, especially for SAT models. For example, in [182], random generation is used to generate random stimuli to test circuits. Some testing problems have also been expressed with CP models, for example because of the need for non-Boolean variables: in [121], the authors define a variability model on continuous variables. They then discretise these variables and sample solutions using RandomSearch. In [20], the Test Suite Reduction problem is tackled with constraint optimisation problems using global constraints. In [183], array constraints are used to handle data structures. Our work brings sampling to these CP models.

Instance generation is also an area where sampling methods are used. In [17], the authors use uniform sampling to generate instances, but as the size of the instances to be generated increases, RandomSearch is used as a more efficient approach. In [190], random generation is enhanced by genetic algorithms to generate interesting instances. In [4], a parameter tuning tool is used to find a value for the parameters of instances such that the generated instance is neither too hard nor too easy to solve. The parameter tuning tool used, `irace`, randomly generates parameters in promising spaces, and updates the random distribution after testing these parameters.

The broad literature on constrained sampling has already been presented in the previous Chapter 4. We would like to point out that our approach is inspired by the works on the hashing constraints presented in the previous chapter, Section 4.2.1. Hashing constraints have recently been used in a CP context for model counting in [41]. These constraints were then used in the sampler LinMod-s [86]. We experiment our approach with the linear modular equality constraints (instead of the `table` constraints) and we com-

pare it to LinMod-s in Section 5.8. In some sense, our approach is also related to the SearchTreeSampler sampler [74] as it enumerates and restricts the allowed values for a subset of variables.

5.2.2 Definitions

In this chapter, we only consider constraint satisfaction problems $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$. A constraint $C \in \mathcal{C}$ is defined by its scope $scp(C)$ (the variables involved in the constraint) and its relation $rel(C)$ (the allowed values for the variables in the scope). In this chapter, we use `table` constraints.

Definition 4 (Table constraint)**.** Given a tuple of r variables $X_{i_1}, \ldots, X_{i_r}$, and a set of tuples $\mathcal{T}$, the table constraint $C = \texttt{table}((X_{i_1}, \ldots, X_{i_r}), \mathcal{T})$ is such that $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, and $rel(C) = \mathcal{T}$.

These constraints allow to directly define the allowed values for the variables. We use `table` constraints as hashing constraints. If a family of hashing constraints divides the possible solutions evenly and independently, it is said to be r-independent.

Definition 20 (r-independence)**.** Let $\mathcal{X}$, and let $\mathcal{H}$ be a family of constraints on all the variables of $\mathcal{X}$. The family $\mathcal{H}$ is r-independent iff for a random $h \in \mathcal{H}$ and for $\sigma_1, \ldots \sigma_r$ instantiations of the variables in $\mathcal{X}$

$$\mathbb{P}\left(\sigma_r \in rel(h) \mid \sigma_1 \in rel(h) \wedge \ldots \wedge \sigma_{r-1} \in rel(h)\right) = \mathbb{P}\left(\sigma_r \in rel(h)\right)$$

This property of a family of hashing constraints is central in the proofs of uniformity of samplers. In [41] a family of hashing constraints, the linear modular equalities, were introduced as extensions to integer variables of XOR constraints.

Definition 22 (Linear Modular Equality)**.** Let p be a prime number. Let $a_1, \ldots, a_n \in \mathbb{F}_p$ be coefficients, $b \in \mathbb{F}_p$ be a constant, and $X_1, \ldots, X_n$ be n variables. A *linear modular equality* is an equality

$$\sum_{i=1}^{n} a_i X_i \equiv b \pmod{p}.$$

The family $\mathcal{H}_{(\text{mod } p)}(n)$ is the set of all linear modular equalities modulo p, i.e.

$$\mathcal{H}_{(\text{mod } p)}(n) = \left\{ \sum_{i=1}^{n} a_i X_i \equiv b \pmod{p} \mid a_1, \ldots, a_n \in \mathbb{F}_p, b \in \mathbb{F}_p \right\}$$

```
1 Function RandomTable(P, v, p)
      Data: A CSP P = ⟨{X_1, ..., X_n}, D, C⟩, v > 0, 0 < p < 1
      Result: A random table constraint
2     T ← {}
3     i_1, ..., i_v ← GetIndices(P, v)
4     foreach (x_{i_1}, ..., x_{i_v}) ∈ ∏_{k=1}^{v} D(X_{i_k}) do
5         if Random() < p then
6             T.add((x_{i_1}, ..., x_{i_v}))
7     return table((X_{i_1}, ..., X_{i_v}), T)
```

Algorithm 5.1: Random table constraint generation algorithm

This family of hashing constraints requires the choice of a prime number p larger than the maximum range of the domains. If such a p is chosen, this family is 2-independent.

RandomSearch During the solving, when no more propagation can be done, the search strategy chooses an uninstantiated variable X and a value v in its domain, and makes the decision $X = v$ (or its negation $X \neq v$). The random search strategy RandomSearch chooses the variable uniformly at random among all the uninstantiated variables, and chooses the value uniformly at random from the domain of the variable. This search strategy makes decisions in constant time, but these decisions can lead to unsatisfiable spaces, hence finding solutions more slowly than with dedicated search strategies.

5.3 TableSampling

We present here a new approach to sample solutions in a CSP. This approach is twofold: first we present a way to generate random table constraints, a key component of the method, and second, we present an algorithm to sample solutions using these generated constraints.

5.3.1 Random Table Constraints

The algorithm for generating random table constraints is presented in Algorithm 5.1. We assume that the following functions are available:

- Random() which returns a random floating point number between 0 and 1,
- GetIndices$(\mathcal{P}, v)$ which returns v indices $i_1, \dots i_v$ such that $|\mathcal{D}(X_{i_k})| \neq 1, 1 \leq k \leq v$ (if there are fewer than v such indices, they are all returned),

- and $\texttt{table}(\mathcal{X}', T)$ which creates a table constraint C such that $scp(C) = \mathcal{X}'$ and $rel(C) = T$.

In addition to the CSP $\mathcal{P}$, the algorithm has two parameters: v the number of variables in the table, and p the probability of adding a tuple to the table. The algorithm first randomly selects v variables from those whose domains are not reduced to a singleton, iterates through all the instantiations of these v variables, and adds each instantiation in the table with probability p. The goal of these tables is to reduce the solution space to a smaller sub-space. The following theorem shows that, on average, the number of solutions to the problem is reduced by a factor p.

Theorem 2. *Let $\mathcal{P}$ be a CSP, and T be a table constraint randomly generated with probability p. Then*

$$\mathbb{E}\left(|Sols\left(\mathcal{P} \wedge T\right)|\right) = p\,|Sols\left(\mathcal{P}\right)|$$

or equivalently, if σ is an instantiation, then

$$\mathbb{P}\left(\sigma \in rel(T)\right) = p\,.$$

Proof. For $\sigma \in Sols\left(\mathcal{P}\right)$, let γ_σ be a random variable equal to 1 if and only if $\sigma \in Sols\left(\mathcal{P} \wedge T\right)$. $\mathbb{P}\left(\gamma_\sigma = 1\right)$ is the probability that σ satisfies T. Let $X_{i_1}, \ldots, X_{i_r}$ be the variables chosen in T. Every instantiation of these variables has been added in the table with probability p, including the instantiation $(\sigma(X_{i_1}), \ldots, \sigma(X_{i_v}))$. This means that σ satisfies the table constraint T with probability p. We thus have $p = \mathbb{P}\left(\gamma_\sigma = 1\right) = \mathbb{E}\left(\gamma_\sigma\right)$. It follows:

$$\begin{aligned}
\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) &= \mathbb{E}\left(\sum_{\sigma \in Sols(\mathcal{P})} \gamma_\sigma\right) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} \mathbb{E}\left(\gamma_\sigma\right) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} p \\
&= p|Sols(\mathcal{P})|
\end{aligned}$$

□

The purpose of Theorem 2 is the following: by adding table constraints, we reduce the size of the solution set by a factor p on average. Since p is a parameter of the algorithm,

```
1  Function TableSampling(P, κ, v, p)
       Data: A CSP P, κ ≥ 2, v > 0, 0 < p < 1
       Result: A solution to the problem P
2      S ← FindSolutions(P, κ)
3      if |S| = 0 then
4          return "No solution"
5      while |S| = 0 ∨ |S| = κ do
6          T ← RandomTable(P, v, p)
7          S ← FindSolutions(P ∧ T, κ)
8          if |S| ≠ 0 then
9              P ← P ∧ T
10     return Random (S)
```

Algorithm 5.2: Sampling algorithm by adding table constraints

we can control how fast the reduction is performed. A low value of p has a higher chance of making the problem inconsistent, but a higher value of p reduces the solution space less.

5.3.2 Sampling Algorithm

The sampling algorithm is presented in Algorithm 5.2. First, we present the helper functions used in this algorithm. The first one is $\textsc{Random}\,(S)$, which returns a random element taken uniformly in S. The second function is $\textsc{FindSolutions}(\mathcal{P}, s)$, which enumerates the solutions of $\mathcal{P}$ until s solutions have been found, and returns them. Note that, if this function returns s solutions, then $|Sols(\mathcal{P})| \geq s$, and if it returns fewer than s solutions, then all the solutions have been found. The depth-first search in constraint solvers makes it easy to implement such a function.

The sampling algorithm works as follows: table constraints are added to the problem to reduce the number of solutions. When there are fewer solutions than a given pivot value, a solution is randomly returned from the remaining solutions. The algorithm is described in details in Algorithm 5.2. There are three parameters in addition to the CSP $\mathcal{P}$. A value κ for the pivot is chosen to limit the number of solutions enumerated in the intermediate problems, as well as the number of variables per table v and the probability p to add a tuple in the table. The algorithm first enumerates κ solutions and stops immediately if there are no solutions, or fewer than κ solutions. If the problem has more than κ solutions, a new table constraint is randomly generated. If the problem with this constraint still has

solutions, the constraint is definitively added to the problem (this is the purpose of the test line 8). The algorithm stops when there are fewer than κ solutions. Finally, one of the remaining solutions is randomly chosen and returned.

The solutions are returned one by one by our approach, similarly to UniGen and for the same reasons: once a solution has been chosen, the tables used to find this solution cannot be kept to choose another one, as this would create dependencies. Thus, to generate multiple *independent* solutions, the algorithm is run several times from scratch. Furthermore, there is no guarantee on the size of the final set, except for the one ensured by line 5, $0 < |S| < \kappa$. In other words, the number of solutions in the final set cannot be fixed. If a user does not mind the bias described above, it is very easy to return the final set directly, and run the algorithm again until the desired number of solutions is found.

5.3.3 Proof of termination

When designing random algorithms, one must be particularly careful about the termination. Here we show that Algorithm 5.2 terminates with probability 1.

We fix values for $\kappa \geq 2, v > 0$ and $0 < p < 1$. The case of the initial problem not being satisfiable is caught at the beginning of the algorithm (line 3).

The following lemmas show that there always exists a table that reduces the number of solutions to the problem without making it inconsistent, and that this table is chosen with a non-zero probability. Without loss of generality, we assume that there are always v variables in the tables. If fewer than v variables are not instantiated, we pick some of the already instantiated variables and use their current values to complete the instantiations.

Lemma 1. *Let $\mathcal{P}$ be a problem with at least two solutions. In our framework, there exists a random table constraint T_0 such that*

$$0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$$

Proof. Let σ_1 and σ_2 two distinct solutions of the problem $\mathcal{P}$. Let i_1 be such that $\sigma_1(X_{i_1}) \neq \sigma_2(X_{i_1})$. Let $i_2, \ldots, i_v$ be other indices such that $|\mathcal{D}(X_{i_k})| \neq 1, 2 \leq k \leq v$. Let us define the table

$$T_0 = \texttt{table}\left((X_{i_1}, \ldots, X_{i_v}), \{(\sigma_1(X_{i_1}), \ldots, \sigma_1(X_{i_v}))\}\right)$$

Then $\sigma_1 \in Sols(\mathcal{P} \wedge T_0)$ so $Sols(\mathcal{P} \wedge T_0) \neq \varnothing$, and $\sigma_2 \notin Sols(\mathcal{P} \wedge T_0)$ so $Sols(\mathcal{P} \wedge T_0) \neq Sols(\mathcal{P})$. Since we add a constraint to $\mathcal{P}$ to build $\mathcal{P} \wedge T_0$, we have $Sols(\mathcal{P} \wedge T_0) \subseteq Sols(\mathcal{P})$.

In the end, we have: $Sols(\mathcal{P} \wedge T_0) \neq \varnothing$, $Sols(\mathcal{P} \wedge T_0) \subseteq Sols(\mathcal{P})$ and $Sols(\mathcal{P} \wedge T_0) \neq Sols(\mathcal{P})$, thus $Sols(\mathcal{P} \wedge T_0) \subset Sols(\mathcal{P})$, so $0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$. □

Lemma 2. *There exists a constant $\rho > 0$, depending only on the initial problem, such that, for T a randomly chosen table constraint with v variables:*

$$\mathbb{P}\left(0 < |Sols(\mathcal{P} \wedge T)| < |Sols(\mathcal{P})|\right) \geq \rho$$

Proof. We know from Lemma 1 that there is at least one table constraint T_0 such that $0 < |Sols(\mathcal{P} \wedge T_0)| < |Sols(\mathcal{P})|$. Let d be the maximum size of the domains of the initial problem. We bound the probability of RandomTable(v, p) picking exactly T_0 (up to ordering of the scope of the constraints). Let T be a random table returned by RandomTable(v, p). We want to bound

$$\begin{aligned}\mathbb{P}(T = T_0) &= \mathbb{P}\left(scp(T) = scp(T_0) \wedge rel(T) = rel(T_0)\right) \\ &= \mathbb{P}\left(scp(T) = scp(T_0)\right) \cdot \mathbb{P}\left(rel(T) = rel(T_0) \mid scp(T) = scp(T_0)\right)\end{aligned}$$

There are $\binom{n}{v}$ ways of choosing the v variables appearing in the table (the order does not matter), so $\mathbb{P}\left(scp(T) = scp(T_0)\right) = 1/\binom{n}{v}$. Let k be the number of tuples in T_0. There are at most d^v possible tuples in total. The probability of choosing any tuple in T_0 and not the others is $p^k(1-p)^{d^v-k}$. Since $k \leq d^v$ we have the lower bound $\mathbb{P}\left(rel(T) = rel(T_0) \mid scp(T) = scp(T_0)\right) \geq p^k(1-p)^{d^v-k} \geq min(p, 1-p)^{d^k}$. By defining $\rho = \frac{1}{\binom{n}{v}} min(p, 1-p)^{d^k}$ we have the desired bound, and $\rho > 0$ because $0 < p < 1$. □

We have proved that during an iteration of the loop, there is a probability strictly greater than 0 of removing solutions without making the problem inconsistent. We can now prove that the algorithm terminates with probability 1. The proof is similar to the one showing that tossing a fair coin until tails comes up ends with probability 1.

Theorem 3. *Algorithm 5.2 terminates with probability 1.*

Proof. For some $k > |Sols(\mathcal{P})| - \kappa$, we want to find an upper bound on the probability that the algorithm has not stopped after k iterations. In some cases, an iteration reduces the number of solutions to the problem without making it inconsistent. There can be at most $|Sols(\mathcal{P})| - \kappa$ such iterations, because the algorithm stops if there are fewer than κ solutions (condition of the while, line 5). For the other iterations, the condition of the while loop ensures that: either the (most recently added) table has made the problem

inconsistent, or it has not reduced the number of solutions. The probability of making the problem inconsistent or not reducing the number of solutions is less than $1 - \rho$, as stated in Lemma 2. Thus, the probability that the algorithm has not stopped after k iterations is less than $(1 - \rho)^{k-|Sols(\mathcal{P})|+\kappa}$. This probability tends to zero as k tends to infinity. This proves that the algorithm stops with probability 1. □

This proof is built with an upper bound, and considers the worst case (when solutions are slowly eliminated), but in practice there is more than one table that satisfies Lemma 1. The solving time in practice will be studied in Section 5.7.

5.3.4 Dichotomic table addition

From early experiments, we noticed a behaviour of the algorithm that led us to create a variant. In fact, there is little chance that the first tables added will make the problem inconsistent. On the contrary, after many iterations, several tables have been added, and it becomes very fast to find the κ solutions (or to prove inconsistency). This is due to the fact that all the previously added tables really restrict the search space and are quickly propagated.

It is possible to modify the algorithm by increasing the number of tables added at each step. This will reduce the number of iterations at the beginning of the algorithm. At the end, it increases the probability of having an inconsistent problem, but as we have seen, it is very fast to prove inconsistency in the last iterations. There is a trade-off between the number of steps of the algorithm and the number of inconsistent problems created. Depending on the problem this variant may or may not be faster than the baseline algorithm.

The exact algorithm is inspired by the unbounded dichotomic search: first, find i such that the value we want to guess is between 2^i and 2^{i+1}, and then, run a usual dichotomic search between 2^i and 2^{i+1}.

The algorithm of dichotomic table addition is presented in Algorithm 5.3, and should replace lines 6 to 9 of Algorithm 5.2. Let τ be the number of tables added in the previous step, we choose $nbTables = 1$ if $\tau = 0$ or $nbTables = 2\tau$ otherwise, and $nbTables$ tables are generated and stored in an array $\mathcal{T}$. The algorithm then enumerates κ solutions to the problem where the tables in $\mathcal{T}$ have been added. If there are no solutions, it deletes half of the constraints in $\mathcal{T}$. The procedure stops when the problem is satisfiable or $|\mathcal{T}| = 0$.

1 **Function** *DichotomicTableAddition*($\mathcal{P}, nbTables, \kappa, v, p$)
  Data: A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $nbTables > 0, \kappa \geq 2, v > 0, 0 < p < 1$
  Result: $\mathcal{P}$ with the new table constraints, and the number of added tables
2 $\mathcal{T} \leftarrow$ array of size $nbTables$
3 **for** $i = 0$ *to* $nbTables - 1$ **do**
4 $\mathcal{T}[i] \leftarrow$ RandomTable($\mathcal{P}, v, p$)
5 $S \leftarrow$ FindSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$)
6 **while** $|S| = 0 \wedge |\mathcal{T}| > 0$ **do**
7 $\mathcal{T} \leftarrow \mathcal{T}[0 : |\mathcal{T}|/2[$
8 $S \leftarrow$ FindSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$)
9 **return** $\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, |\mathcal{T}|$

Algorithm 5.3: Dichotomic addition of tables variation

Theorem 3 can be extended to the case of the dichotomic table addition, because line 6 in Algorithm 5.3 ensures that the problem does not become inconsistent.

This variant of the algorithm has comparable running times to the baseline algorithm (as seen in Section 5.7.1). It performs better on some instances, and worse on others. A user can try both variants on some instances before running the full sampling, in order to choose the best variant for their application.

5.4 Discussion

In this section, we discuss the algorithmic choices we have made in Algorithm 5.2, compare it more closely with Meel's approach, and extend our approach to other hashing constraints.

5.4.1 Quality of Table's Division

In the proof of Theorem 2, the random variables $(\gamma_\sigma)_{\sigma \in Sols(\mathcal{P})}$ are not independent. For example, let σ_1 and σ_2 be two solutions to the problem that only differ in one variable X, then

$$\mathbb{P}(\gamma_{\sigma_2} = 1 \mid \gamma_{\sigma_1} = 1) = \mathbb{P}(X \in scp(T)) \cdot p + \mathbb{P}(X \notin scp(T)) \tag{5.1}$$

Indeed, if the variable X appears in T, then σ_2 is be kept with probability p, but if X is not in the scope of T, then σ_2 is always be kept. If the table does not have all the variables in its scope, then it may not split the clusters of solutions that take the same values on

multiple variables. This notion of independence is central to Meel's approaches [82] to show the uniformity of the sampling. In contrast to this approach, our sampling is not uniform. We choose to have tables of a controlled size for sake of efficiency.

Formula 5.1, which shows the non-independence, also shows that increasing the number of variables in the table makes the random variables γ_σ more independent, thus bringing the whole sampling process closer to uniformity. Tables containing all the variables of the problem would make the random variables γ_σ fully independent (i.e. n-independent), since in this case $\mathbb{P}(X \notin scp(T)) = 0$. This would give a theoretical guarantee of sampling, but is impossible to generate in practice.

5.4.2 Comparison with ApproxMC

In his thesis [82], Kuldeep Singh Meel presented an algorithm to count (ApproxMC) and then to sample solutions of SAT formulas (UniGen) based on adding of XOR constraints to the problem. Their counting algorithm ApproxMC adds multiple XOR constraints to the SAT formula until there are less than a given number of solutions, and then extrapolates the total number of solutions. Running this function several times gives a probably approximately correct (PAC) counter, i.e. given two parameters $0 < \epsilon$ and $0 < \delta < 1$, if c is the value returned by the algorithm with the parameters ϵ and δ on the formula $\mathcal{F}$, then

$$\mathbb{P}(|Sols(\mathcal{F})|/(1+\epsilon) \leq c \leq |Sols(\mathcal{F})|(1+\epsilon) \geq 1-\delta$$

He then uses this counter to get an almost uniform sampler (the probability of sampling is close to the uniform by a factor ϵ, where ϵ that can be chosen).

Our approach is inspired by ApproxMC but differs in that we have traded the proven uniformity for a faster algorithm. The constraint used to reduce the solution space is not an XOR constraint (or its extension to CP, a linear modular equality constraint), but a `table` constraint. A `table` constraint with few variables allows for a propagation closer to the root of the search tree, whereas a linear modular equality only propagates close to the bottom of the search tree. The 2-uniformity of the linear modular equalities is also their downfall: it ensures that the solutions satisfying the constraint are well distributed throughout solution space. Therefore, it is not possible to use this constraint to efficiently propagate and cut large sub-spaces of the search space.

The algorithm to get a sample has also been simplified to what is strictly necessary.

ApproxMC has to run the algorithm several times to get a proven model counter, and then use that model count to sample a solution. Our algorithm is applied only once and will always return a solution (whereas UniGen may not return a solution).

5.4.3 Using Different Constraints

Algorithms 5.2 and 5.3 have been presented using the random tables, but they can actually be used with any constraint that divides the space. In fact, we can replace the line 6 in Algorithm 5.2 (or the line 4 in Algorithm 5.3) by the creation of any constraint that we want. In our experiments we tested this sampling algorithm with randomly generated linear modular equalities (with randomly picked coefficients). This set of hashing constraints ensures the strong property of 2-independence of the presence of solutions. We call this algorithm (our approach using linear modular equalities) LinModEq and evaluate the quality of the randomness and the running time in the experiments. We expect to get a better sampling distribution using these constraints, and eventually a uniform sampling.

5.4.4 Influence of the Parameters

Three parameters must be chosen to run the algorithm. We discuss here their influence on the running time and on the quality of the randomness.

- As seen in the previous subsection, increasing the number of variables in the tables should improve the randomness, but will also exponentially increase the number of tuples in the table, with a negative impact on the running time.
- Reducing the probability of adding a tuple in a table should improve the running time because the tables will be smaller, so the propagation will be faster, and the number of tables added will be lower because the problem will be reduced more quickly.
- The influence of the pivot on the running time is unclear. A higher pivot means that more solutions have to be enumerated at each step, but it also means that the algorithm stops after adding fewer constraints.

These hypotheses are verified experimentally in Section 5.6.2.

5.5 Experiments Methodology

This section presents the methodology used to carry out the experiments. First we present the details of the implementation, and then we present the benchmarks, their characteristics, and the reason for using each one of them. The approach is independent of the constraints of the problem, so we were able to apply it to different problems without being limited by the presence or absence of a constraint.

5.5.1 Implementation

The code is available online [1], along with the benchmarks used and all the scripts used to generate the figures presented in this chapter. TableSampling has now been integrated in `choco-solver` since the version 4.10.9.

TableSampling

The implementation was done in Java 11 using the constraint solver `choco-solver` version 4.10.6 [46]. It is possible to create a model directly in Java using the `choco-solver` library, or by passing a file in the FlatZinc format (generated from the MiniZinc format). If no strategy is defined in the FlatZinc file, the solver's default strategy is used (`dom/Wdeg` [12] and `lastConflict` [32]).

A technical variation has been made by adding a propagation step before creating a table (before line 6 of Algorithm 5.2). This avoids enumerating some tuples that would be immediately deleted by propagation. This small variation is evaluated in Section 5.7.1.

In the following, TableSampling refers to the algorithm using DichotomicTableAddition.

LinModEq

The LinModEq approach is Algorithm 5.2, where we replaced the random table constraints with random linear modular equalities. For the implementation of the propagator of the linear modular equality system, we reused the implementation by [41], which is already available in Java. Their implementation is based on the `minicp` solver, modified to use belief propagation [2]. We have adapted their linear modular equality propagator to work in the `choco-solver`'s framework.

1. `https://github.com/MathieuVavrille/tableSampling`
2. `https://github.com/PesantGilles/MiniCPBP`

To ensure that our implementation is not flawed compared to the original implementation, we tested it on the same benchmark as the one used in [41]. We observe the same behaviour, although there are differences due to the underlying CP solver. To enumerate all the solutions (without any linear modular equality), `choco-solver` is an order of magnitude faster than `minicp`. In fact, according to the official website, `minicp` "is not focused on efficiency but rather on readability to convey the concepts as clearly as possible".

Despite the gap in global performance, we observe the same behaviour as [41] when adding linear modular equalities (and by using the propagator designed for a system of linear modular equalities): increasing the number of equalities reduces the running time.

5.5.2 Preliminary Benchmark

First, we use a small benchmark to perform extensive experiments, in order to calibrate the settings of the algorithm. Its purpose is to evaluate the randomness of our approach and the impact of the parameters in order to extract a generic set of parameters to use as a baseline. The evaluation of the randomness has to be done on small problems due to the computational cost of the χ^2 test.

This benchmark consists of three problems, which are describe in detail below. The results of the evaluation on this preliminary benchmark are presented in Section 5.6. The running time comparison with RandomSearch and LinModEq will be done with a harder benchmark.

Problems

The following problems have been chosen for their reasonable number of solutions (to apply the χ^2 test), and for their relevance. The computation time is small enough to allow extensive experiments, and we use the results to calibrate the parameters of our method.

N-queens The first problem is the N-queens problem, which consists of placing N queens on an $N \times N$ chessboard in such a way that no queen attacks any other queen (queens attack in all 8 directions, as far as possible). We implemented it using the classical model with N variables with domain $[1, N]$, and binary disequality constraints (the same model as the one used in [41]). We use the 9-queens instance, which has 352 solutions.

On Call Rostering This problem models the rostering system used in partular by health workers. This instance is available in the MiniZinc benchmarks [3] and contains different types of constraint, such as linear constraints, global constraints `count`, absolute values, implications and table constraints. Many datasets are available but only the smallest (`4s-10d.dzn`) was used here. This is an optimisation problem (minimisation), so it was necessary to transform this problem into a satisfaction problem by limiting the objective function. The optimal value is 1:

- There are 136 solutions with $obj \leq 1$
- There are 2,099 solutions with $obj \leq 2$
- There are over 10,000 solutions with $obj \leq 3$

By randomising the solutions, the solver can be used as a decision support tool for the planners (giving them several plans to compare) and brings a form of equity between the workers. Indeed, oriented search methods could favour some employees at the expenses of others.

Feature Model This is a problem of software management problem that helps to decide on the order of implementation of software features. The instance is specified in the MiniZinc format in [184] using the data in [186]. Again, this is an optimisation problem (maximisation), the optimal value is 20,222. We add the constraint $obj \geq 17,738$ to make it a satisfaction problem with 95 solutions.

Evaluation of uniformity

Evaluating the randomness of a system is a difficult task. In fact, random systems can take surprising values without being biased: for example, a fair coin will occasionally land heads ten times in a row. The chi-squared (or χ^2) test is a classical method for comparing the result of a random experiment with an expected probability distribution. It derives from a convergence result of the χ^2 law, given in [181] and recalled here. Let Y be a random variable on a finite set, which takes the value k with probability p_k for $1 \leq k \leq d$. Let $Y_1, \ldots, Y_n$ be independent random variables with the same law as Y. Let $N_n^{(k)}$ be the number of variables $Y_i, 1 \leq i \leq n$ equal to k.

Theorem 4 ([181]). *As n tends to infinity, the cumulative distribution of the random*

3. `https://github.com/MiniZinc/minizinc-benchmarks/tree/master/on-call-rostering`

variable

$$Z_n = \sum_{k=1}^{d} \frac{\left(N_n^{(k)} - n \cdot p_k\right)^2}{n \cdot p_k}$$

tends to the cumulative distribution of the law of the χ^2 *with* $(d-1)$ *degrees of freedom (noted* χ^2_{d-1}*).*

The χ^2 test boils down to picking values at random, assuming that they follow the law of Y, computing the experimental value z_n^{exp} of Z_n, and computing the probability (called the p-value)

$$\mathbb{P}(Z_n \geq z_n^{exp}) \approx \mathbb{P}(\chi^2_{d-1} \geq z_n^{exp})$$

If this probability is close to zero, then, having a more extreme result than the one obtained is very unlikely. This means that the hypothesis that the experimental values follow the same law as Y can be confidently rejected. Conversely, if the p-value stays close to one, we can confidently assume that the experimental values follow the same law as Y. Here, we are interested in the uniform distribution, i.e. $\forall k \in \{1, \ldots, d\}, p_k = 1/d$.

Experimentally, knowing the number $nbSols$ of solutions to a problem (and numbering these solutions), $nbSamples$ samples are drawn and count the number of occurrences $nbOcc_i$ of each solution $i \in \{1, \ldots, nbSols\}$. We compute the value of the variable

$$z_{exp} = \sum_{k=1}^{nbSols} \frac{(nbOcc_k - nbSamples/nbSols)^2}{nSamples/nbSols}$$

and then the p-value of the test [4] (i.e. the probability that the χ^2 law takes a more extreme value than z_{exp}). This p-value gives a numerical measure of the quality of the randomness. More precisely, a large number of samples are drawn (more than the number of solutions) and the evolution of the p-value is plotted as a function of the number of samples. In order to perform this test, we need to know the number of solutions $nbSols$ and to sample $nbSols$ solutions multiple times, so the evaluation of the randomness can only be done on small instances.

As an example to understand the evolution of the p-value, let us consider a problem with two solutions, and suppose that our sampling method is biased towards the first solution, returning it with probability 0.6. After taking 10 samples, the experimental solution distribution might be $(6, 4)$ (i.e. 6 times the first solution, and 4 times the second).

4. We use the library "Apache Commons Mathematics Library" (`https://commons.apache.org/proper/commons-math/`) for the probability computations

Knowing that we have only taken 10 samples, no one can be sure that the distribution is not uniform. This leads to a high p-value (here the p-value is 0.527). After doing 100 samples, the experimental solution distribution might be $(62, 38)$. At this point, it is becoming unlikely that a uniform distribution could produce such a distribution, but still not impossible. The p-value is getting closer to 0 (here the p-value is 0.0164). After doing 1000 samples, the experimental solution distribution might be $(587, 413)$. Now it is almost impossible to get this distribution from a uniform distribution. The p-value will be extremely close to 0 (here the p-value is $3.7 \cdot 10^{-8}$).

We plot the evolution of the p-value as a function of the number of samples. The non-uniform approaches have a p-value that tends to 0, and the approaches that give a uniform distribution will have a p-value that tends to 1. The rate at which the p-value tends to 0 also gives information about the *distance* to the uniform distribution. In the previous example, if the distribution was $(0.8, 0.2)$, then after sampling 100 solutions the experimental distribution may be $(81, 19)$, hence the p-value would already be extremely close to 0 (in fact the p-value is $5.6 \cdot 10^{-10}$).

So the p-value allows to rank non-uniform approaches. An approach whose p-value tends to 0 more slowly is "more uniform" (but still not exactly uniform) than an approach whose p-value tends to 0 more quickly.

5.5.3 MiniZinc Challenge Benchmark

To evaluate the performance of our approach, we also performed experiments on a second larger benchmark with harder problems from the MiniZinc challenge [5]. The MiniZinc challenge is a yearly solver competition on a large, diverse benchmark. To evaluate our approach, we created a benchmark based on the problems from the 2016 to 2021 challenge.

The challenge contains very hard instances and sets a time limit of 20 minutes. We decided to keep this 20 minutes time limit for our computations. For many optimisation problems, no solver was able to prove that the solutions found (if any) were optimal. We restricted the benchmark to problems where `choco-solver` was able to find and prove the optimal value. We transformed all the optimisation problems into satisfaction problems by fixing the objective function to its optimal value.

We recall that if we choose a parameter $\kappa > |Sols(\mathcal{P})|$, the TableSampling approach boils down to enumerating all the solutions and returning one at random. This

5. `https://www.minizinc.org/challenge.html`

is not at all interesting for testing our approach as it will never add any constraint (either table constraint or linear modular equality). In the experiments, we chose $\kappa = 16$ (see section5.6.2), thus we restricted the benchmark to problems where we were able to enumerate more than 15 solutions before the timeout of 20 minutes.

The approaches are random, so the running time is impacted by this randomness. To limit these random factors, we run each approach 10 times on each instance and average the total time. If an approach times out on one of the runs, we record this as a timeout of the approach on that instance (we are no longer able to average the time).

In summary, this is how the benchmark was built:

- we start with all the instances of the MiniZinc challenge from 2016 to 2021;
- we remove the optimisation problems where we could not find the optimal value in less than 20 minutes. We then transform them into satisfaction problems by fixing the objective to the optimal value;
- we remove all the problems with than 16 solutions (enumerated in less than 20 minutes);
- we run each approach 10 times on each instance and record the average running time.

The final benchmark contains 82 instances.

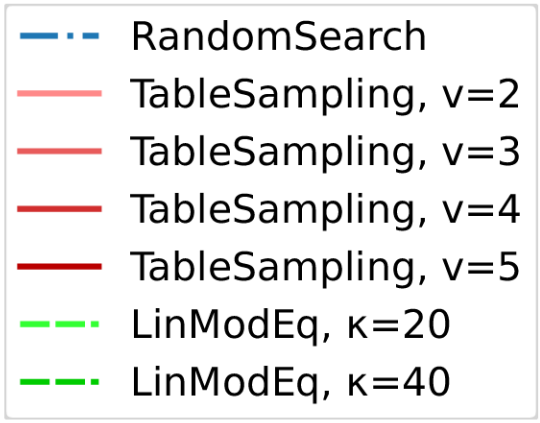
5.6 Preliminary Experiments

This section presents the results of the preliminary experiments carried out to evaluate the impact of the parameters. RandomSearch, TableSampling and LinModEq were run to sample the problems multiple times. Different sets of parameters (for κ, v, and p) were used for TableSampling and LinModEq. Figures 5.1, 5.2, 5.3 and 5.5 show some results that illustrate the behaviour of the approaches.

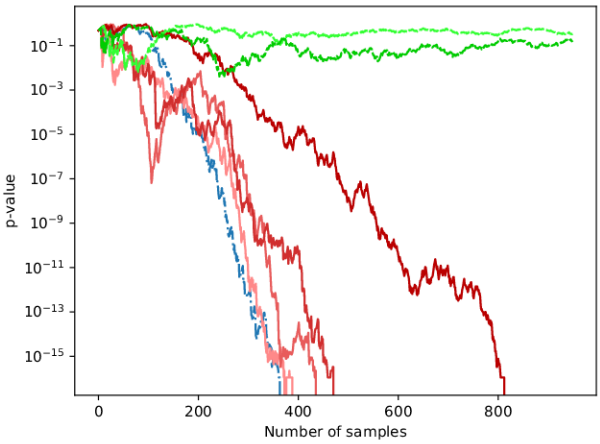
Remark. *The figures show the p-value on a logarithmic scale, because it tends to 0. Also, since the computations are done using floating point representation, a p-value less than* 10^{-16} *is considered to be equal to 0.*

5.6.1 Quality of the Randomness

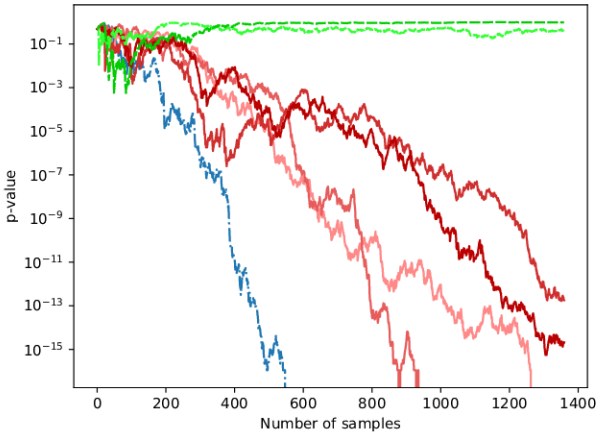
The first goal of the experiments is to evaluate the quality of the randomness, i.e. to know if the solutions are randomly and uniformly sampled. The following results show



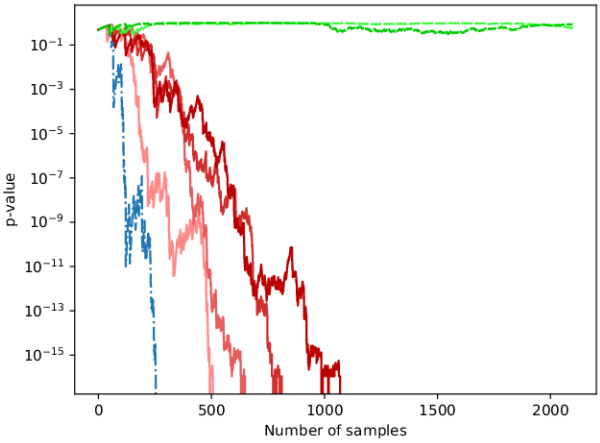
(a) Legend for the sub-figures.



(b) Feature Model instance.



(c) On Call Rostering instance with the constraint $obj \leq 1$.



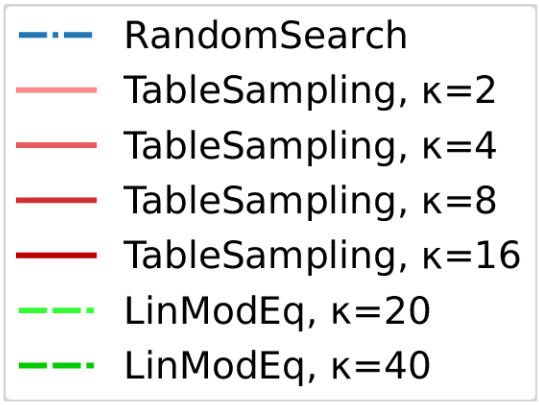
(d) On Call Rostering instance with the constraint $obj \leq 2$.

Figure 5.1 – Evolution of the p-value on different problems, with $\kappa = 16, p = 1/8$, and different values for v.

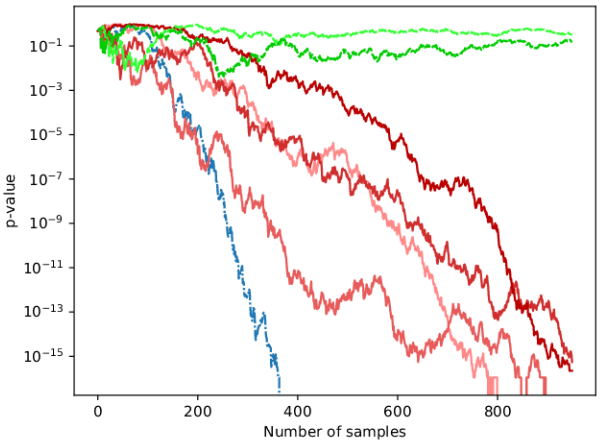
that even if the solutions are not sampled uniformly, TableSampling is *more uniform* than the RandomSearch strategy.

The graphs for different problems are shown in Figures 5.1, 5.2, and 5.3. On each graph, there is one line (in dash-dotted blue) for the p-value of RandomSearch, several plain lines for TableSampling with different parameters (in shades of red), and several dashed lines for LinModEq with different values for κ (in shades of green). A p-value tending towards 0 means that the sampling is not uniform, and a p-value tending towards 1 means that the sampling is uniform.

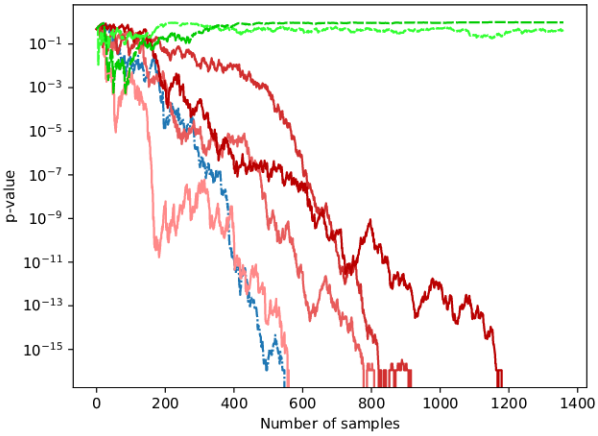
The first remarkable fact is that LinModEq, i.e. our approach using the linear modular equalities instead of the table constraints, is uniform. For the two values of κ that



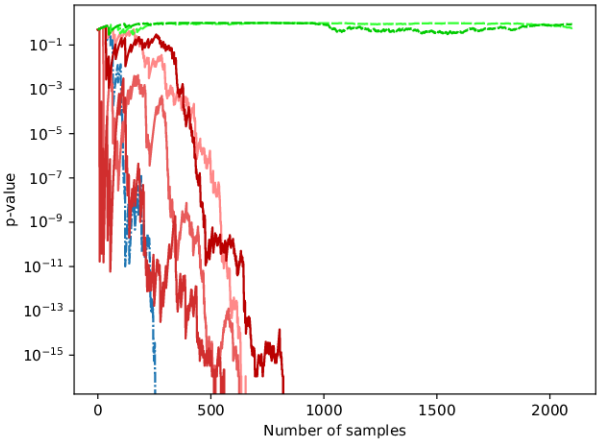
(a) Legend for the sub-figures.



(b) Feature Model instance.



(c) On Call Rostering instance with the constraint $obj \leq 1$.



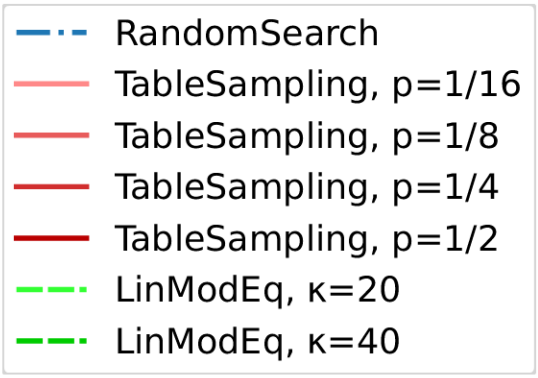
(d) On Call Rostering instance with the constraint $obj \leq 2$

Figure 5.2 – Evolution of the p-value on different problems, with $v = 5, p = 1/2$, and different values for κ.

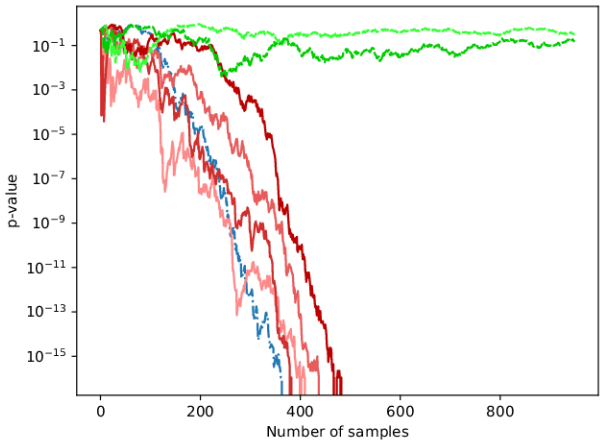
are plotted, the p-value always tends to 1. This was expected as discussed in section 5.4.3.

The second observation, also expected, is that RandomSearch is never uniform. We have seen on a very simple example that this search strategy does not sample uniformly, and in these more structured problems, it is even more obvious.

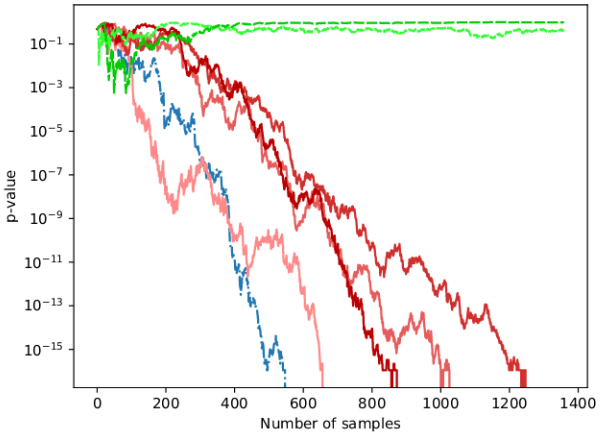
The behaviour of our approach lies between these two approaches. For most problems, the sampling is not uniform. However, it is always closer to the uniform distribution than RandomSearch. This can be seen from the fact that the p-value for TableSampling tends to 0 more slowly than for RandomSearch (no matter what the set of parameters is used). On the 9-queens problem (in Figure 5.4) the p-value even tends to 1, which means that our approach samples solutions uniformly, even though RandomSearch is



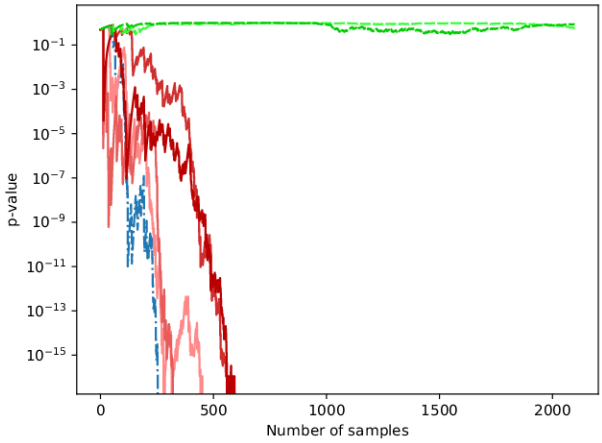
(a) Legend for the sub-figures.



(b) Feature Model instance.



(c) On Call Rostering instance with the constraint $obj \leq 1$.



(d) On Call Rostering instance with the constraint $obj \leq 2$

Figure 5.3 – Evolution of the p-value on different problems, with $\kappa = 8, v = 3$, and different values for p.

still not uniform. We believe that this is due to the structure of the solution space, since the N-queens problem is a very structured problem with many symmetries. Thus, it is likely that the solutions are well distributed in the search space.

These experiments show that our approach is really beneficial for the quality of the randomness compared to the basic random strategy. By using table constraints as hashing constraints, one can improve the randomness, but the sampling may not be uniform. By using linear modular equalities, it is possible to sample uniformly from the solution set, at the disadvantage of an increased running time, as linear modular equalities are harder to propagate, and they propagate slower than table constraints.

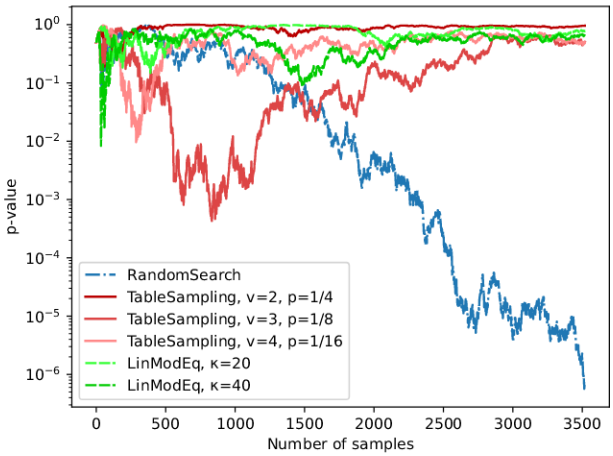


Figure 5.4 – Evolution of the p-value on the 9-queens problem with $\kappa = 8$ and different parameters for v and p.

5.6.2 Impact of the Parameters

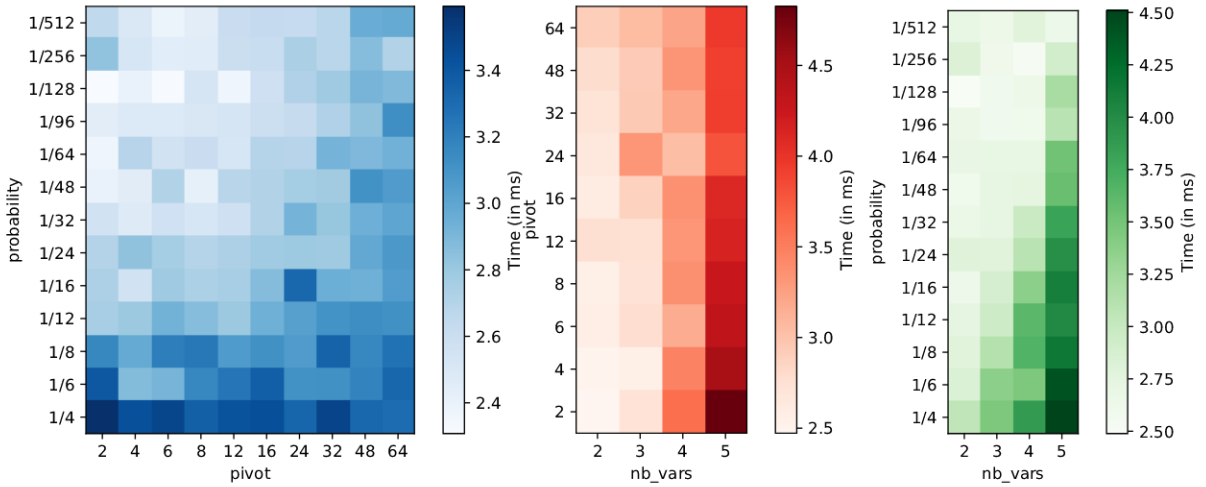
Our approach has the advantage of being parametric. It allows the users to choose the most appropriate parameters their application. However, this requires a good knowledge of the impact of the parameters on the running time and the quality of the randomness. Here we verify the hypotheses made in Section 5.4.4.

We ran experiments by changing the parameters, and evaluating the evolution of the p-value and the running time with these different parameter sets. We use the previous evaluation of the p-value, and for the running time figure 5.5 shows heat maps of the running times. In these heat maps, one parameter is fixed, and we vary the others two. The darker the colour, the longer it took to sample a solution.

Impact of the Number of Variables

First we vary the number of variables used in the generated tables. Figure 5.1 shows the evolution of the p-value on the instances with parameters $\kappa = 16$ and $p = 1/8$. We remark that as the number of variables in the table increases, the p-value tends to zero more slowly, meaning that the sampling is closer to uniformity. As we noted earlier, this is due to a better independence in the probability that two solutions will satisfy the table constraints (see section 5.4.1).

The evolution of the running time can be seen in the heat maps 5.5c and 5.5b. We can clearly see that increasing the number of variables increases the running time. This behaviour is easily explained, because increasing the number of variables exponentially



(a) $v = 3$ (b) $p = 1/16$ (c) $\kappa = 16$

Figure 5.5 – Heat maps of the time to sample one solution, by fixing different parameters, on the On Call Rostering instance with the constraint $obj \leq 3$.

increases the number of tuples in the tables.

Impact of the Pivot

In Figure 5.2 we vary the pivot on the different instances, with the parameters $v = 5$ and $p = 1/2$. Here we observe that increasing the pivot improves the randomness. In fact, when the pivot is high, a lot of solutions are enumerated at each step, and in the end a random solution is picked among a lot of other solutions, leading to a better randomness. The extreme case is the perfect (but costly) sampling process, when the pivot is higher than the number of solutions: the algorithm then simply performs an enumeration of all the solutions, and returns a (perfectly uniform) random solution.

The evolution of the running time can be seen in the heat maps 5.5b and 5.5a. From these results, there is no clear impact of the pivot on the running time.

Impact of the Probability

Figure 5.3 shows the p-value for different values of the table probability p, and with the parameters $\kappa = 8$ and $v = 3$. There is no clear influence of the probability of adding tuples in the tables on the quality of the randomness.

However, the probability does have an impact on the running time. We see on the heat maps 5.5c and 5.5a that decreasing the probability decreases the running time. A small

probability allows to have smaller tables, so the solution space is reduced faster. Thus the algorithm converges faster to a small set of solution (smaller than κ). There is no point in reducing the probability too much, because at some point the average table will be empty. One has to find a trade-off for the probability p, and it depends on the average size of the variable domains. For variables with large domains, the probability must be small in order to keep the tables tractable. On the other hand, tables that hold only on Boolean variables need a higher probability, to avoid empty tables.

Base set of parameters

The number of variables in the tables should be chosen as a trade-off between the desired quality of randomness and the running time. It will also depend on the application: instances with big domains may require smaller v to avoid too large tables (for example, $v = 4$ for domains of size 100 means enumerating 10^8 tuples). From our experiments, we suggest the default parameter values: $\kappa = 16$ and $p = 1/16$. We choose to have $p = 1/\kappa$, because it reduces the chances of having inconsistencies after adding a table: we know that the problem has more than κ solutions. Thus, after adding a table with probability $1/\kappa$, there will be more than one solution on average.

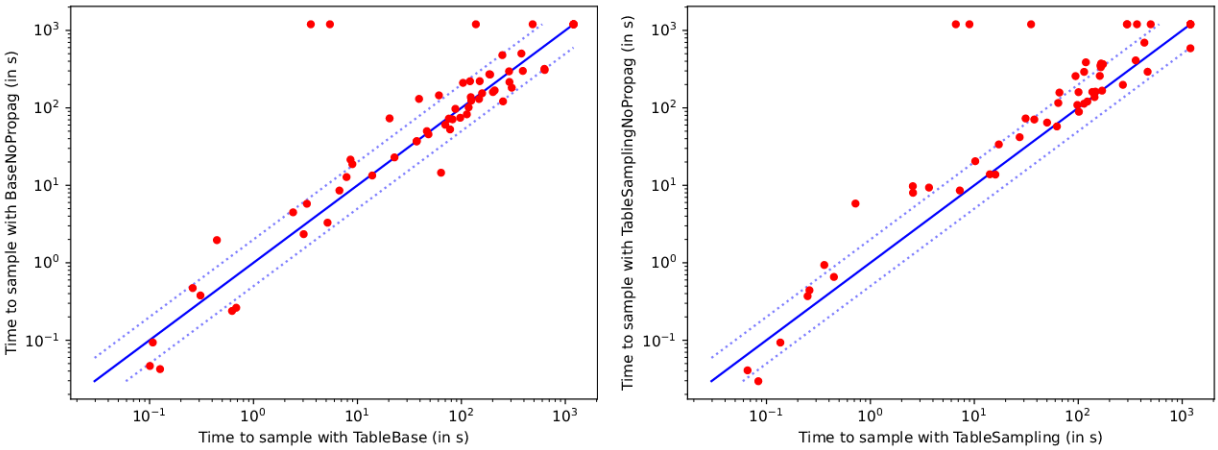
5.7 MiniZinc Challenge Experiments

We present here the running time results on the MiniZinc challenge benchmark, as presented in Section 5.5.3. The raw results are given in Appendix B. The experiments were performed with the parameters $\kappa = 16, v = 2$ and $p = 16$.

5.7.1 Difference between the variants

In this section we study the running time of variants of the base algorithm. We note BaseNoPropag the base algorithm presented in Algorithm 5.2. When the propagation step (introduced in Section 5.5.1) is added, we note the approach TableBase. Then there is the dichotomic variant of the algorithm. With propagation we call it TableSampling, and without propagation we call it TableSamplingNoPropag.

In Figure 5.6 we show a scatter plot of the running time for each instance and we compare the approaches with and without the propagation step. In Figure 5.6a we cannot see a significative difference between the algorithm with, or without the propagation step



(a) Base variant (b) Dichotomic variant

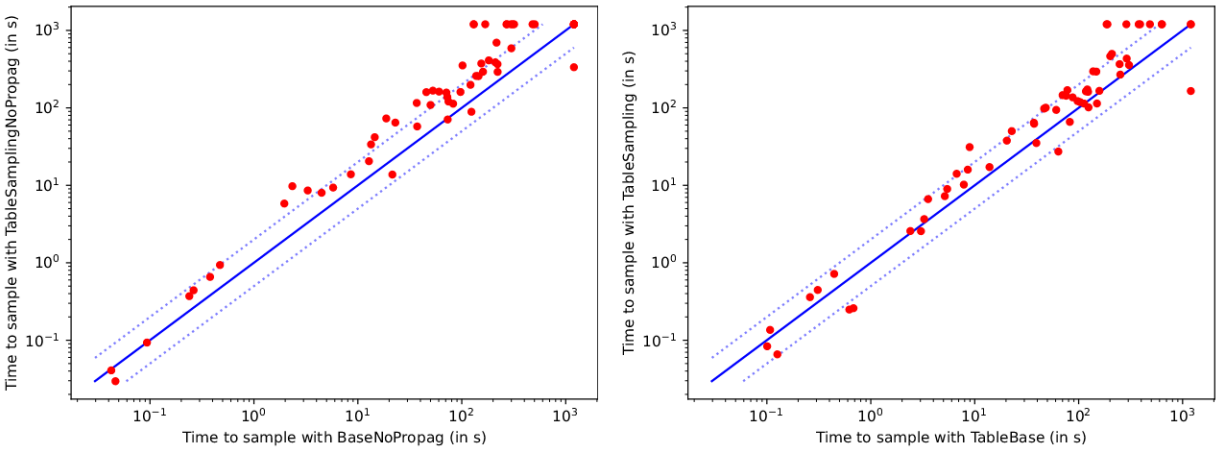
Figure 5.6 – Scatter plot of the time to sample one solution for the base and dichotomic variants, compared to their version without propagation. Each point is an instance. The plain blue line is the $x = y$ line, and the dotted lines are the $x = 2y$ and $x = y/2$.

on the base algorithm. However, using the dichotomic variation, the propagation improves the running time. In fact, without the propagation step, in some instances, due to the size of the tables generated, a Java heap space error was raised (i.e. too much memory was used).

In Figure 5.7 we show the impact of using the dichotomic variant of the algorithm. Without the propagation step, in Figure 5.7a we can see that TableSamplingNoPropag is worse than the base algorithm BaseNoPropag for the same reason as before: the tables generated are too large and costly to generate. With the propagation step, the two variants are close to each other. Depending on the application, a user may test both approaches on a smaller instance to see if one or the other is faster.

5.7.2 Timeouts

The first interesting measure of the running time of an approach is the number of instances solved. Here we look at the number of instances that the approaches were able to sample in less than 20 minutes, out of a total of 83 instances. Figure 5.8 shows two plots of the number of instances sampled by each approach. Figure 5.8a simply shows the histogram of the number of sampled instances. Our approach, TableSampling, was able to sample the most instances: the base variant sampled 58 instances and the dichotomic variant (TableSampling) sampled 51 instances. RandomSearch was able



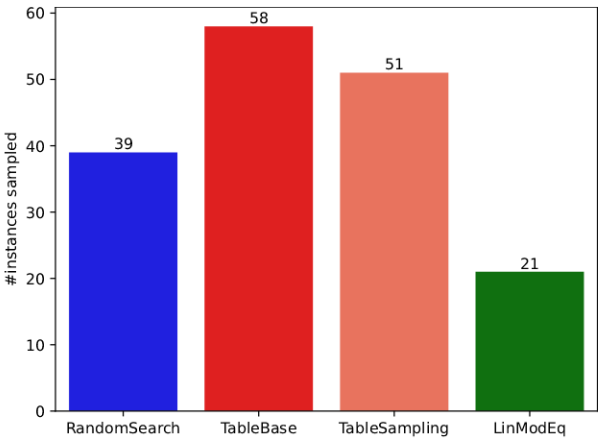
(a) Without propagation (b) With propagation

Figure 5.7 – Scatter plot of the time to sample one solution with the base variant (with and without propagation), compared to the dichotomic version (with and without propagation). Each point is an instance. The plain blue line is the $x = y$ line, and the dotted lines are the $x = 2y$ and $x = y/2$.

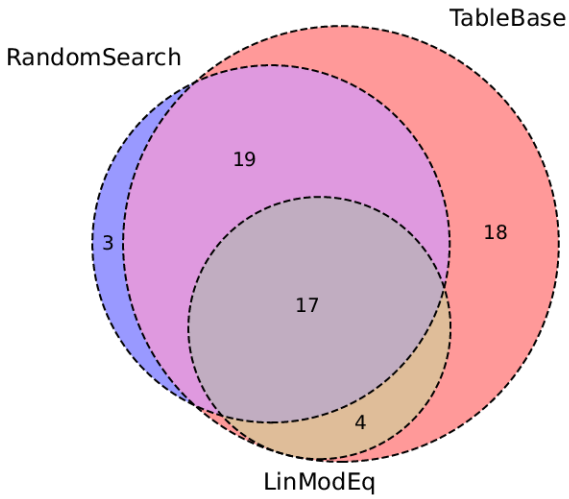
to sample 39 instances, and LinModEq is behind with only 21 sampled instances. Recall that the main goal of our approach was to ensure a good randomness at a low cost. We have already shown experimentally that TableSampling improves the randomness over RandomSearch. In fact, we see that we are able to sample more instances, so we are also competitive in terms of running time.

To understand more precisely the differences between the approaches, a Venn diagram of how many instances are solved by each approach is plotted in Figure 5.8b. We show TableBase, as LinModEq does not use the dichotomic variant. The three circles represent the three approaches, and the different intersections between these circles represent the number of instances sampled by one or more approaches. For example, the subset containing the 4 (in yellow) means that exactly 4 instances were sampled by only TableSampling and LinModEq. There are different takeaways from this Venn diagram:

- TableBase can sample all the instances that LinModEq samples (the circle of LinModEq is contained in the circle of TableBase). This was expected because the linear modular equality constraints used in LinModEq are more expensive to propagate than table constraints. This is the price to pay for a better randomness. When using the dichotomic variant, there is one instance that is sampled by LinModEq and not TableSampling.
- 19 instances were sampled by TableSampling but not RandomSearch, whereas



(a) Histogram of the number of sampled instances per algorithm.



(b) Venn diagram of the number of sampled instances. For example the magenta subset contains 19 instances solved only by TableSampling and RandomSearch, but not LinModEq.

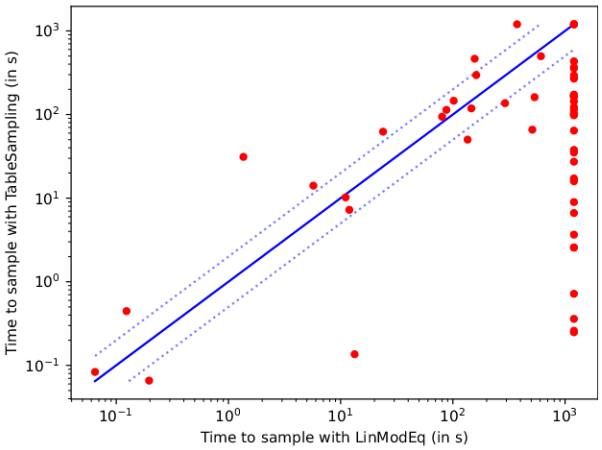
Figure 5.8 – Number of instances sampled by each approach.

only 7 instances were sampled by RandomSearch but not TableSampling. We believe this is due to the ability of TableSampling to keep good search heuristics.

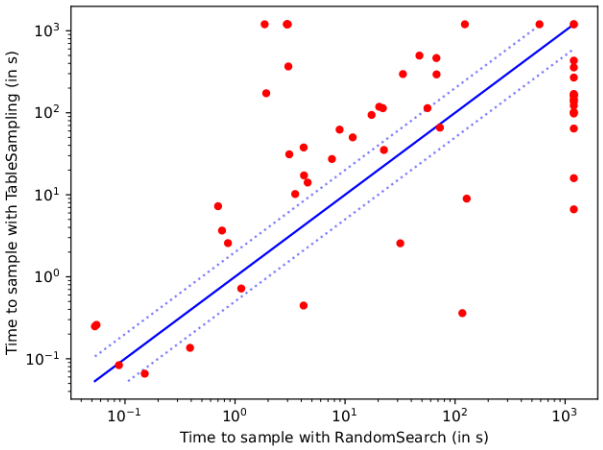
5.7.3 Running Time

The timeouts give a first insight into the comparison of the approaches, but we can also look at the running time on the instances that were sampled without timeouts. Figures 5.9a and 5.9b are scatter plots of the sampling time for each instance. In both figures TableSampling is on the y-axis, so if an instance is above the diagonal, it means that TableSampling was slower to sample it compared to the other approach. Figure 5.9c is another representation of the running times where for each approach we plot the number of instances sampled faster than a given time (the x-axis).

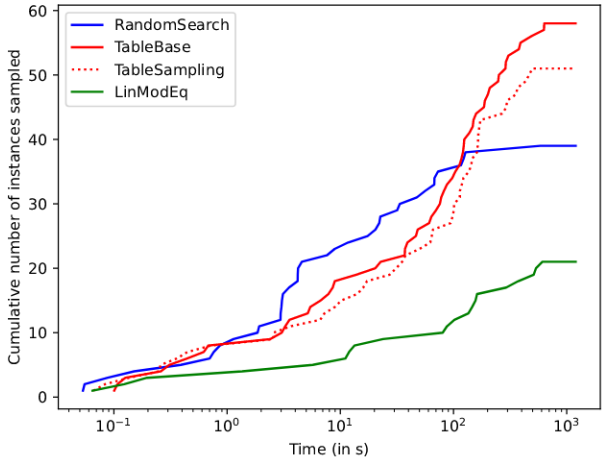
Comparing LinModEq and TableSampling, there are a lot of variations in the running time. The plot mostly show that a lot of instances are sampled by TableSampling (and sometimes very fast), and not by LinModEq. This was to be expected, because the base algorithm is the same, but linear modular equalities are harder to propagate, and propagate less (and later) than table constraints.



(a) Scatter plot of the time to sample one solution for LinModEq versus TableSampling. Each point is an instance.



(b) Scatter plot of the time to sample one solution for RandomSearch versus TableSampling. Each point is an instance.



(c) Plot of the cumulative number of instances sampled in less than a given time.

Figure 5.9 – Plots of the running time of each approach.

The comparison between RandomSearch and TableSampling is very interesting. On the instances that were sampled by both approaches, RandomSearch is often faster than TableSampling. However, as mentioned above, many instances are only sampled by TableSampling (the 22 instances where RandomSearch took 1200s to sample, i.e. timed out), and these instances seem to be among the harder instances (TableSampling takes from one second to a few minutes to sample them). This behaviour can be explained by the fact that RandomSearch is a search strategy, i.e. it cannot coexist with other, efficient heuristics. Using a random search strategy on hard instances is likely to result in bad branching decisions, leading to a lot of time spent in unsatisfiable subtrees of

the search tree. On the contrary, our approach does not require any modification of the heuristics, if efficient ones exist. Thus, TableSampling can benefit from all the dedicated (or black box) search strategies designed in solvers.

On the other hand, the 3 instances solved by RandomSearch and not by TableSampling identify a limit of our algorithm. These instances have variables with large domains. For example, in the `zephyrus` instances, some variables have a domain of size 4097. With $v = 2$ in our experiments, this can lead to an enumeration of more than 8 million tuples. These large domains are a limit of our approach because the creation and the propagation of the random tables will be costly. In this situation, the design of other random table generation algorithms tables could be interesting. For example, it is possible to generate tables with a fixed number of tuples. It would also be possible to use the variables with a small domain first, or to have a v that changes during the resolution, depending on the size of the domains.

5.8 Comparison with LinMod-s

Due to the parallel and overlapping work in [41, 86] we want to clarify the timeline of the publications. The article [41] on linear modular equalities was published in CPAIOR 2021, with submissions in January and the conference in July. At the same time, we submitted the TableSampling conference article [1] to the CP conference 2021, with submission in May, and the conference in October 2021. We extended the conference article in December 2021 using linear modular equalities and submitted it to the journal Constraint. This extended article was accepted and published in July 2022 [2]. At the same time, the article [86] was published in the CPAIOR 2022 conference, with submission in early December and the conference in June. This section is new compared to our journal publication [2], to present the comparison between TableSampling and the recent CP sampler LinMod-s [86].

In the previous chapter we postponed the presentation of the recent sampler using linear modular equalities and inequalities LinMod-s [86]. We present it here, and we then compare it with the experiments we carried out by using the linear modular equalities instead of the `table` constraints.

5.8.1 LinMod-s

In [86] the authors use linear modular equalities and inequalities. We have already introduced the family $\mathcal{H}_{(\mathrm{mod}\ p)}(n)$ of linear modular equalities. We now introduce the family $\mathcal{H}^{\leq}_{(\mathrm{mod}\ p)}(n,c)$ of linear modular inequalities.

Definition 25 (Linear Modular Inequality)**.** Let p be a prime number, n an integer, and $c \in \{1,\ldots,p-2\}$, the family $\mathcal{H}^{\leq}_{(\mathrm{mod}\ p)}(n,c)$ is defined as

$$\mathcal{H}^{\leq}_{(\mathrm{mod}\ p)}(n,c) = \left\{ \sum_{i=1}^{n} a_i X_i + b \leq c \mid a_1,\ldots,a_n \in \mathbb{F}_p, b \in \mathbb{F}_p \right\}.$$

Remark. *In this definition b is an offset. Without this offset, the instantiation $(0,\ldots,0)$ would always be a solution. Also, the value c cannot be equal to $p-1$ because it would not restrict the space.*

Linear modular inequalities can be represented by a disjunction of linear modular equalities. The propagation algorithm for a system of linear modular inequalities is based on this disjunction and is presented in [86]. On average, linear modular equalities with a prime number p reduce by a factor p the number of solutions of the problem (Property 4). The linear modular inequalities retain more solutions because more values are allowed for the linear sum.

Property 6. *Let $X_1,\ldots,X_n$ be n variables, and σ be an instantiation on these n variables. Let h be a random linear modular inequality constraint from $\mathcal{H}^{\leq}_{(\mathrm{mod}\ p)}(n,c)$, then*

$$\mathbb{P}(\sigma \in rel(h)) = \frac{c+1}{p}.$$

Compared to equalities, inequalities can be used to reduce fewer solutions (by choosing a large value for c).

LinMod-s, the sampling algorithm presented in [86], uses the families $\mathcal{H}_{(\mathrm{mod}\ p)}(n)$ and $\mathcal{H}^{\leq}_{(\mathrm{mod}\ p)}(n,\cdot)$ to add linear modular equalities and inequalities to the problem. Algorithm 5.4 presents the base blocks of LinMod-s. It takes as parameters the problem $\mathcal{P}$ and a reduction factor λ, and returns on average close to $\lambda\,|Sols(\mathcal{P})|$ solutions. This algorithm is substantially different to our Algorithm 5.2.

To sample solutions, the algorithm first calls an auxiliary function Partition, which chooses how many equalities and inequalities to add to the problem, and the bounds of

```
1 Function LinMod-s(P, λ)
      Data: A CSP P with n variables, 0 < λ < 1
      Result: A set of solutions to the problem P
2     m, F ← Partition(λ, p);                          // λ ≈ 1/p^m ∏_{c∈F} (c+1)/p
3     for i ∈ {1, ..., m} do
4         h ← Random(H_{(mod p)}(n));
5         P ← P ∧ h;
6     for c ∈ F do
7         h ← Random(H^{≤}_{(mod p)}(n, f));
8         P ← P ∧ h;
9     return Sols(P);
```

Algorithm 5.4: LinMod-s: sampling with linear modular equalities and inequalities

the inequalities. The output values are m the number of equalities, and a set F of bounds for the inequalities. These values are chosen by the function so that the added constraints approximately reduce the number of solutions by a factor λ. Each of the m equalities reduces the number of solutions by a factor $1/p$, and for each inequality factor $c \in F$, the inequality generated from $\mathcal{H}^{\leq}_{(\text{mod } p)}(n, c)$ reduces the number of solutions by a factor $(c+1)/p$. In total, Partition reduces the number of solutions by a factor $\frac{1}{p^m} \prod_{c \in F} \frac{c+1}{p}$.

The algorithm then adds the desired number m of random linear modular equalities and inequalities using the selected values in F. It then returns all the remaining solutions of the constrained problem.

5.8.2 Comparison to TableSampling

LinMod-s takes a different approach by sampling multiple solutions at once. If a given number of solutions is desired, the parameter λ is difficult to estimate. It requires to know an approximation of the total number of solutions. In practice the number of solutions to the problem will not be known in advance, so several values of λ should be tested.

On the other hand, LinMod-s can be easily extended to an approximate model counter. On average, the number of solutions is reduced by a factor λ, so one can easily estimate the total number of solutions by running the sampler several times, counting how many solutions it returns, and multiplying that number by λ.

The experiments of the authors of LinMod-s confirm our experiments. They show

experimentally that their approach is uniform using the χ^2 test, as we did. They also run TableSampling and got similar results to ours. In one set of instances of their experiments (the synthetic instances), TableSampling is experimentally close to the uniform distribution.

Overall, LinMod-s is very close to our algorithm using linear modular equality constraints. The main differences are in the parameters. LinMod-s reduces the number of solutions by a factor λ, but our approach finds by itself how many constraints should be added to have few solutions left. The authors also remark that the linear modular constraints only propagate values late in the search tree. Using `table` constraints allows to easily prune large parts of the search space, speeding up the solving.

5.9 Conclusion

We presented an algorithm that uses `table` constraints to randomly sample solutions of a problem. Experiments show that our algorithm provides a reasonably good quality of randomness, while keeping the computation time tractable. The most important feature of our method is that it does not require any change to the solver settings or the model.

Our approach is lightweight, because it uses the solver as a black box. Moreover, the `table` constraints offer a wide range of possibilities to tweak the solving process according to the user's needs. For example, by playing with the probabilities for certain tuples to be selected, one can orient the sampling in certain subspaces, depending on the user's needs. This allows us to tackle randomisation with any given distribution, not necessarily uniform. On the same idea, reducing the probability of tuples contained in previously found solutions would induce a diversified search.

Exploiting the random reduction of the search space leads to other promising ideas. For example, portfolio algorithms runs several solving processes in parallel, which ideally all search in different subspaces. Feeding the processes with randomly reduced search spaces would force them to explore different subspaces without any biases.

This chapter also began to question the links between sampling (or diversity) and search strategies. On the one hand, RandomSearch randomises the algorithm, but not uniformly. On the other hand, dedicated search strategies make strong choices about the spaces searched, and are strongly biased towards some spaces, but they can find solutions quickly. In the next part of this thesis, we use search strategies to find diverse solution sets.

Conclusion

In this part, we have focused on samplers, i.e. algorithms that generate a random solution satisfying the constraints. It is not difficult to generate random solutions, but it is difficult to ensure some properties of the generation. There is often a trade-off between the guarantees of the sampler (uniformity, near-uniformity) and the running time.

We have presented TableSampling, a sampler dedicated to constraint programming problems. It uses `table` constraints as hash constraints. We decided not to focus on uniformity, but rather on running time, to return solutions quickly. This sampler uses a CP solver as a black-box, so it will benefit from improvements in solvers, such as more constraints or a better running time.

In the following part, we generate diverse solutions to constrained problems using a different approach: search strategies. We design two randomised search strategies tailored to the problems we study. These search strategies allow to search in interesting sub-spaces to find diverse solutions. We also show that sampling uniformly from the solution space does not necessarily generate diverse solutions.

Part III

Search strategies

Chapter 6

Pattern Mining

This chapter comes from an unpublished collaboration with Samir Loudni, Arnold Hien and Albrecht Zimmermann on a pattern mining problem. It is a continuation of Hien's work on diversity in pattern mining in [109]. In this contribution, Samir Loudni and Albrecht Zimmermann helped with most of the related works section 6.6 (and the pattern mining definitions), and Arnold Hien helped by running the experiments (to run other state-of-the-art approaches). I contributed everything else, i.e. the design of the search strategy, the implementation (I started from Hien's implementation in [109] and improved it), the analysis and the presentation of the results, and the writing.

6.1 Introduction

Recently, several data mining problems have been expressed in Constraint Programming (CP), allowing users to define complex queries using high-level languages [97, 104, 108, 110, 115]. CP solvers are modular, so queries can be refined without revising the solving process, unlike dedicated pattern mining algorithms. Additional constraints can easily be added to suit the needs of a user [113]. For example, a *pruning function* such as the *total transaction price* (a bound on a weighted sum on the pattern) as seen in [96] is natively handled by CP solvers. More recently, Hien et al. [109] proposed a global constraint to mine patterns of interest, ensuring that the results are diverse with respect to the Jaccard index, a classical metric in pattern mining. The authors had to relax the problem to deal with the non-monotonicity of the Jaccard index, which limits the efficiency of the constraint.

However, databases are often huge, and the number of patterns found by the solvers can be far too large to be useful. Both human experts and downstream algorithms need small sets of patterns to work with. One of the most classical constraints added on the patterns is the frequency. The problem of frequent pattern mining was introduced in [96] for the task of association rule mining. It allows to find many interesting relationships

between data. The frequent patterns shown to a user should also be *diverse* to avoid the repetition of information, which wastes the expert's time or leads algorithms astray. A classic approach is to first mine a large set of patterns, and to then select a good subset. However, the infamous *pattern explosion* leads to very large results that are difficult to post-process, especially on dense or large databases.

In this chapter we use search strategies, which are classically designed to improve the efficiency of solvers, as a way to enforce diversity in frequent constrained pattern mining. We propose OrientedSearch, a new strategy, and an associated scoring function, to orient the search towards diverse solution spaces. We measure diversity using the Jaccard index, but our approach can use any diversity measure (monotonous or not). We have experimented our approach on sparse and dense databases. The experiments show that using random search strategies (RandomSearch or the proposed OrientedSearch) significantly improves the diversity of the returned patterns compared to other state-of-the-art approaches. The first solutions returned by OrientedSearch are already very diverse. However, when many patterns are desired, the computation of the score in OrientedSearch can become too expensive. In this case, RandomSearch offers a great diversity and is often the fastest approach.

Outline

This chapter first defines in Section 6.2 the closed frequent pattern mining task, and the diversity problem. Section 6.3 presents our contribution and discusses the design choices. Section 6.4 presents the experimental methodology and Section 6.5 presents the results of the experiments. Section 6.6 discusses related work.

6.2 Background

We first define the pattern mining framework, and adapt the diversity definitions.

6.2.1 Pattern Mining

The pattern mining task takes a database as an input.

Definition 26 (Database)**.** Let $\mathcal{I}$ be a set of n items. We call a *transaction* a subset of $\mathcal{I}$. A *database* is a bag (or multiset) of transactions. A *pattern* (or *itemset*) is a non-empty subset of $\mathcal{I}$.

The number of patterns grows exponentially in the number of items (there are $2^{|\mathcal{I}|} - 1$ patterns). Pattern mining searches for *interesting* patterns. The measure of interestingness depends on the user, but a well-known pattern mining task is *frequent itemset mining* [96]: this is the problem we will focus on.

Definition 27 (Frequent Itemset Mining)**.** The *cover* of an itemset P in D is the (multi)set of transactions in which it occurs, *i.e.*

$$\mathcal{V}_D(P) = \{t \in D \mid P \subseteq t\}\,.$$

In the following, the database will always be fixed, so we just write $\mathcal{V}(P)$ for $\mathcal{V}_D(P)$. The *support* of an itemset P is the cardinality of its cover:

$$sup(P) = |\mathcal{V}(P)|\,.$$

Given a threshold θ, a pattern P is said to be *frequent* if its support is greater than or equal to θ, i.e. $|\mathcal{V}(P)| \geq \theta$. The task of *frequent itemset mining* is to compute all the *frequent itemsets*.

The frequent pattern mining task returns patterns that cover many transactions. However, these patterns may contain redundancy, for example if two patterns have the same cover (i.e. are included in the same transactions). To avoid this redundancy, the notion of *closed* patterns was introduced in [102, 117].

Definition 28 (Closed pattern)**.** A pattern P is said to be closed iff there is no $Q \supseteq P$ such that $sup(P) = sup(Q)$, i.e. P is maximal with respect to set inclusion among the itemsets with the same support.

By restricting the problem to closed itemsets, the itemsets returned will not contain any redundancy. In this chapter we restrict ourselves to closed and frequent itemsets.

Example. *Table 6.1 shows an example of a database. This database contains five items A, B, C, D, E and four transactions. We simplify the notation of the transactions and patterns: for example $t_2 = \{A, B, C\}$ is simplified as $t_2 = ABC$. Given a pattern, the cover is the transactions that contain the pattern. For example, the pattern CE is covered by the transactions t_3 and t_4, so $\mathcal{V}(CE) = \{t_3, t_4\}$.*

We take a threshold of $\theta = 2$ for the frequent patterns. In this case, the pattern AE is not frequent because it only covers the transaction t_1, so $sup(AE) \ngeq \theta$. However, the pattern containing only A is frequent because it covers the transactions t_1 and t_2.

	A	B	C	D	E
t_1	*	*			*
t_2	*	*	*		
t_3		*	*	*	*
t_4		*	*		*

Table 6.1 – Example of database with five items (ABCDE) and 4 transactions (t_1, t_2, t_3, t_4).

Closed patterns are patterns that cannot be extended (by adding another item) without decreasing its cover. For example the frequent pattern A is not closed because the pattern AB also covers the transactions t_1 and t_2 ($\mathcal{V}(AB) = \mathcal{V}(A)$). However, the pattern AB is closed (and frequent) because for every other item $X \in \{C, D, E\}$, $sup(AB) \neq sup(ABX)$. In total there are six frequent closed itemsets in this database with the threshold $\theta = 2$:$\{AB, B, BC, BCE, BE, CE\}$

A CP model for mining frequent closed itemsets was proposed in [115], which successfully encoded both the closeness relation and the frequency into a global constraint called `closedPattern`. It uses a vector of Boolean variables $(X_1, \ldots, X_{|\mathcal{I}|})$ to represent itemsets, where X_i represents the presence of the item $i \in \mathcal{I}$ in the itemset. Given a (partial) instantiation σ the pattern associated with σ is $P = \{i \in \mathcal{I} \mid \sigma(X_i) = 1\}$.

Definition 29 (`closedPattern` constraint [115])**.** Let $(X_1, \ldots, X_{|\mathcal{I}|})$ be a vector of Boolean variables, θ a support threshold and D a database. Let σ be an instantiation, and $P = \{i \in \mathcal{I} \mid \sigma(X_i) = 1\}$ the associated pattern. The constraint $\texttt{closedPattern}_{D,\theta}(X_1, \ldots, X_{|\mathcal{I}|})$ holds iff P is a closed frequent itemset w.r.t. the threshold θ.

The `closedPattern` constraint merges the two constraints (frequent and closed pattern). This constraint has later been improved by `coverSize` [119] to allow the threshold to be a variable. The base model contains only the `closedPattern` constraint, but other constraints can easily be added. In the following, we will denote as a solution any frequent closed itemset.

6.2.2 Solution Diversity

Finding a set of diverse itemsets is an important task in data mining. Several measures have been proposed to measure the diversity of itemsets. In this paper, we consider the

Jaccard index as a measure of similarity on sets, and use it to quantify the overlap of the covers of itemsets.

Definition 30 (Jaccard index)**.** Let P and Q be two itemsets. The *Jaccard index* is defined as

$$J_{\mathcal{V}}(P,Q) = \frac{|\mathcal{V}(P) \cap \mathcal{V}(Q)|}{|\mathcal{V}(P) \cup \mathcal{V}(Q)|}$$

Note that $J_{\mathcal{V}}$ has values in $[0,1]$. Also, it is a similarity measure and not a distance measure, i.e. to ensure a high diversity between pairs of itemsets, a small Jaccard is desired.

The diversity problem has been defined and studied in [24]. `MaxDiverseKSet` is the problem of finding the most diverse set of k patterns. `MostDistant` is a greedy approximation of `MaxDiverseKSet`. We recall here its definition adapted to the Jaccard index.

Definition 12' (`MostDistant` for Pattern Mining)**.** Let $\mathcal{H} \subset Sols(\mathcal{P})$ be a set of closed and frequent patterns. `MostDistant`$(\mathcal{H})$ is the problem of finding the pattern $\widetilde{P}$ that is most distant from all the patterns in $\mathcal{H}$, i.e.

$$\widetilde{P} = \underset{P \in Sols(\mathcal{P})}{\operatorname{argmin}} \max_{P' \in \mathcal{H}} J_{\mathcal{V}}(P,P')\,.$$

The `MostDistant` problem consists in finding the solution that minimises the Jaccard index to the previously found solutions. This chapter presents new ways to approximate this problem, by using search strategies tailored for pattern mining problems.

The ClosedDiv constraint

Recently, Hien et al. have proposed in [109] to add a diversity constraint to the basic closed pattern mining model. Diversity is controlled by a threshold on the Jaccard similarity. This constraint maintains a history of previously found patterns and ensures that the newly mined pattern is diverse with respect to all itemsets in the history.

Definition 31 (Maximum Diversity Constraint)**.** Let $\mathcal{H}$ be a history of patterns, j_{max} a bound on the maximum allowed Jaccard. Let σ be an instantiation, and P be the associated pattern. The *maximum diversity constraint* $\texttt{div}_{J_{\mathcal{V}}}(P, \mathcal{H}, j_{max})$ ensures that P is diverse with respect to the $\mathcal{H}$ and j_{max}, i.e.

$$\texttt{div}_{J_{\mathcal{V}}}(P, \mathcal{H}, j_{max}) \Leftrightarrow \forall H \in \mathcal{H}, J_{\mathcal{V}}(P,H) \leq j_{max}$$

To solve the `MostDistant` problem, j_{max} can be minimised. However, in [109] the authors decided to fix this value at 0.05. As search progresses, each time a solution is found, it is added to $\mathcal{H}$. The constraint is thus modified, and may prune a larger part of the search space. This approach is closely related to dominance programming [114, 116], where each time a solution is found, a *dominance blocking constraint* is added to the model to remove dominated solutions.

Since the Jaccard index has no monotonicity property, in [109] the authors had to relax the constraint. They proposed an anti-monotonic lower bound relaxation that allows to prune non-diverse itemsets during search. This was integrated through the global constraint ClosedDiv. However, the proposed approach provides no guarantees on the actual Jaccard index between the returned solutions (the actual value of the Jaccard is not checked when a solution is found). Furthermore, the number of solutions returned cannot be specified: the solving process stops when the search space is exhaustively explored.

6.3 New Diversification Strategy for Mining

ClosedDiv is a constraint, and as such works at the propagation level. To ensure diversity it prunes parts of the search space that contain only non-diverse patterns.

In this article, we stay within the CP framework, and instead orient the search towards diverse patterns by defining new dedicated search strategies. In this way, we can produce diverse patterns without modifying the model or the internal structure of the solver. Users can thus still refine the model, for example by adding new constraints, depending on their application.

6.3.1 OrientedSearch Strategy

In CP, search strategies lack insight into the distance between the solutions. The rationale behind our new strategy, called OrientedSearch, is to use the Jaccard distance to choose items that will induce solutions that are diverse from the previously found solutions.

At a given step of the solving process, some variables are instantiated and some others are not. The pattern in construction can be retrieved by taking all the variables that are instantiated to 1: $\mathcal{X}^+ = \{i \in \mathcal{I} | X_i = 1\}$. When the solver needs to make a decision, it calls the search strategy. The strategy thus has access to the list of uninstantiated variables,

1 **Function** *OrientedSearch*$(\mathcal{P}, \mathcal{H})$
  Data: A CSP $\mathcal{P} = \langle (X_1, \ldots, X_n), \mathcal{D}, \mathcal{C} \rangle$, a history $\mathcal{H}$
  Result: A decision to perform in the search
2 $\epsilon \leftarrow 10^{-8}$
3 $\mathcal{X}^+ = \{i \in \mathcal{I} \mid X_i = 1\}$
4 $W \leftarrow$ array of size n (indexed from 1)
5 **for** $i = 1$ *to* n **do**
6 **if** $\mathcal{D}(X_i) = \{0, 1\}$ **then**
7 $W[i] \leftarrow 1/(hs(i, \mathcal{X}^+, \mathcal{H}) + \epsilon)$
8 **else** $W[i] \leftarrow 0$
9 $X \leftarrow$ Random $(X_1, \ldots, X_n, W)$
10 **return** Decision$(X = 1)$

Algorithm 6.1: Computation of the decision of OrientedSearch

the ones on which the decision will be made. We rank each uninstantiated variable X_i (associated to the item i) according to a score computed using the Jaccard index with the previously found solutions. We propose the following score:

Definition 32 (History score)**.** Given a pattern $\mathcal{X}^+$ in construction, an item i whose associated variables X_i are uninstantiated, and a history $\mathcal{H}$ of solutions, we define the *history score hs* as

$$hs(i, \mathcal{X}^+, \mathcal{H}) = \max_{H \in \mathcal{H}} J_{\mathcal{V}}(\mathcal{X}^+ \cup \{i\}, H)$$

This score has two important parts. The computed values are the Jaccard indices between $\mathcal{X}^+ \cup \{i\}$ and the solutions of the history. When making the decision, it is preferable to choose an item such that $\mathcal{X}^+ \cup \{i\}$ will be diverse from the solutions of the history. Then all these values have to be aggregated into a single score. We used the max aggregator, which allows to consider the worst Jaccard of all the ones computed, ensuring a minimum diversity. For example, if the score is 0.04, it ensures that the Jaccard distance between $\mathcal{X}^+ \cup \{i\}$ and *all* the solutions of $\mathcal{H}$ is less than 0.04.

We propose to bias a random distribution to choose variables (i.e. items) that will be diverse from the previously found solutions. We use the history score to weight the distribution. The pseudocode is given in Algorithm 6.1.

The current pattern $\mathcal{X}^+$ is extracted from the variables. An array is created to store the distribution that will be used to pick the variable. Instantiated variables are given a weight of 0 (in line 8), as we must not choose them. For each uninstantiated variable, a weight is computed as follows. Recall that a "good" value for the Jaccard index is close to

0. To bias the random distribution towards small values, we have to invert the computed history score. The actual weight used in the random distribution is $1/(hs(i, \mathcal{X}^+, \mathcal{H}) + \epsilon)$ (in line 7). The addition of ϵ is here to avoid dividing by 0. Taking a small enough value for ϵ ensures a deterministic choice when the history score is 0. Finally, a variable is randomly chosen with respect to the weights in W in $\textsc{Random}\,(X_1, \ldots, X_n, W)$ [192], i.e. with probability $W[i]/\sum_j W[j]$. This variable is set to 1 by the decision, adding the item to the current pattern.

6.3.2 Complexity

When designing a search strategy, there is a trade-off between a simple but fast variable selection criterion and a more complicated one at the cost of a longer running time. Our strategy OrientedSearch needs to perform several computations to make an insightful decision. At every decision, for each uninstantiated variable, the Jaccard indices with the solutions in the history are computed. The complexity of computing of the Jaccard index, $J^C(D)$, depends on the size of the database D and the patterns involved. The complexity for a decision of the strategy OrientedSearch is then $\mathcal{O}\left(|\mathcal{X}| \cdot |\mathcal{H}| \cdot J^C(\mathcal{D})\right)$. At the root node (i.e. the first decision of each restart), the history score can be computed incrementally, and only the new solution needs to be processed, giving a complexity for the root decision of $\mathcal{O}\left(|\mathcal{X}| \cdot J^C(D)\right)$.

For the task of diverse pattern mining, this complexity is not an issue. Only a few patterns are desired by a user, so the history score is quickly computed for each decision. However, in the experiments we generate many patterns, and in some cases the computation of the decision becomes expensive.

6.3.3 Best of Random and Greedy Algorithms

The OrientedSearch strategy presented is a fusion of two ideas, keeping the best of both worlds. On the one hand, a fully random strategy RandomSearch (choosing the variables with equal probability) provides coverage of the solution space, but lacks insight into the distance. On the other hand, a fully greedy algorithm (choosing the variable that minimises the history score) provides diversity, but may get stuck in a local optimum, preventing from searching in the whole solution space. Our strategy allows to keep the strength without the weaknesses, finding diverse solutions in the whole solution space.

6.4 Experimental Methodology

This section presents the methodology used in the experiments, the results of which are presented in Section 6.5. The experimental evaluation focuses on the achieved diversity and running times. For the diversity, we look at the global diversity (of the many solutions returned), but we also look at the first solutions returned. Only a few solutions are presented to a user, so in a diversity setting, only the first few solutions are used.

6.4.1 Databases

We consider a wide range of real-world databases coming from the CP4IM repository [1]. The database statistics for each are shown in Table 6.2. For each database we show the number of items, teh number of transactions, and the density (relative number of 1s). We have selected databases of various sizes and densities. We have taken some of the largest and most dense databases, such as `hepatitis` and `chess`. Others, such as `T10I4D100K` and `retail`, are very sparse (resp. 1% and 0.06%). Different support thresholds were chosen for each database. These thresholds were chosen to be as low as possible, while still allowing the processing to finish within 24 hours for .

6.4.2 Comparison with Other Approaches

We compare our diversity strategy with several state-of-the-art approaches. We are interested in approaches that can tackle the problem we are studying, i.e. *frequent* and *closed* pattern mining. The first approaches that we tested are based on CP solvers, and can therefore handle frequency constraints and more. In the CP framework, constraints can easily be added to tackle a specific problem.

- We compared to ClosedDiv (already presented in Section 6.2.2) with a maximum diversity threshold $j_{max} = 0.05$. ClosedDiv is the only approach, among all the tested ones, where the number of solutions cannot be fixed. We use the number of solutions returned by ClosedDiv to fix the number of solutions returned by the other approaches.
- We used the naive random strategy RandomSearch, a search strategy that chooses the variable to branch on uniformly at random (as opposed to our weighted random distribution).

1. `https://dtai.cs.kuleuven.be/CP4IM/datasets/`

- PostHoc [28] is a two-step *ad hoc* algorithm to find a given number k of diverse solutions. We used, as proposed in [28], a random approach for the first step to generate $K = 2k$ solutions (we use two variants, RandomSearch and OrientedSearch), and a greedy approach for the second step to extract the k solutions.

We also compared with an approach that is not based on CP, but that can enforce the frequency constraint.

- Flexics [107] is a sampling method, we use the EFlexics variant, using the Eclat solver for pattern mining. It splits the search space into cells using random XOR constraints, and then draws a certain number of patterns from these cells. EFlexics is based on a specialised search procedure, Eclat, and is therefore not as generic as the CP approaches as it can only sample frequent and closed patterns, other constraints are not supported.

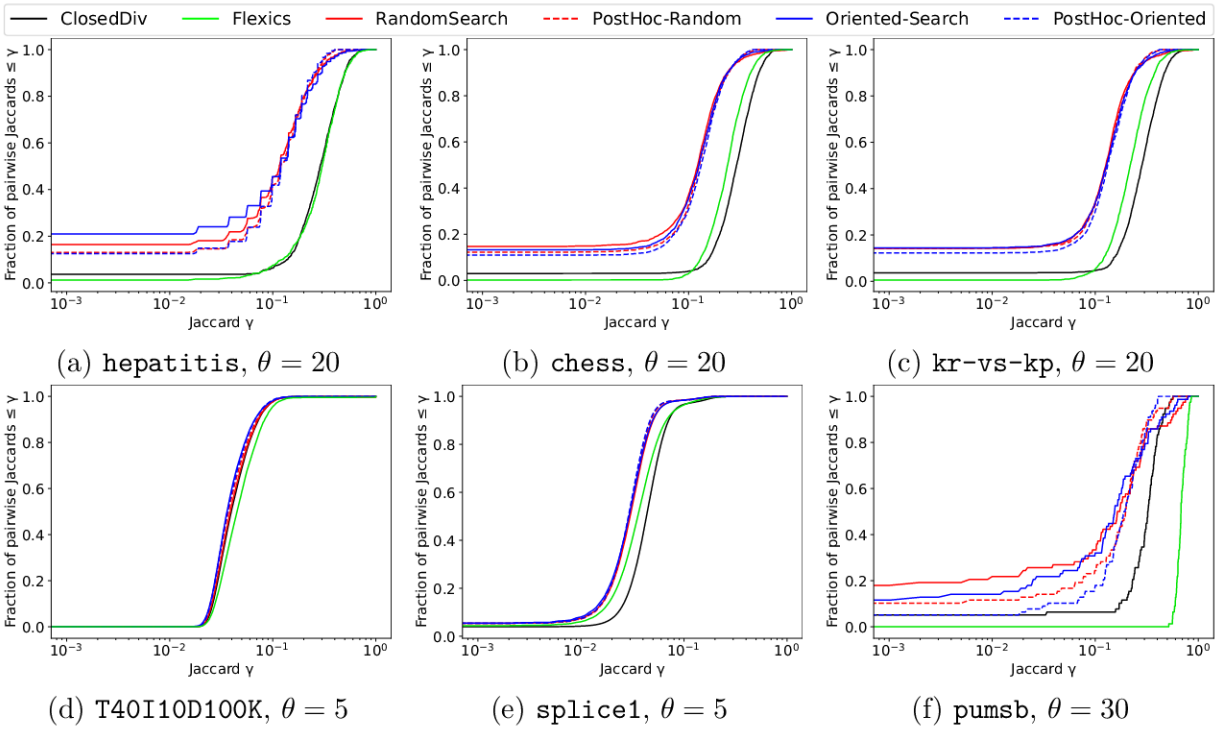
6.4.3 Implementation

Our implementation is available online. [2] It is built upon the implementation of ClosedDiv in [109], which uses the CP solver `choco-solver-4.10.7` [46]. Implementations of Flexics and ClosedDiv were made available to us by the original authors. Flexics is implemented in Scala, the others are implemented in Java. All experiments were conducted as single-threaded runs on AMD Opteron 6174 (2.2GHz) processors with 256 GB of RAM and a 24-hour time limit.

6.5 Experimental Results

The quality of a set of solutions depends on the user's needs. In this section we show two different ways to evaluate the diversity of a set of solutions. In Section 6.5.1 we show plots of the diversity of the whole solution set. In Section 6.5.2 we show plots of the average Jaccard on the first solutions returned by the approaches. [3] We also plot the running time in Section 6.5.3. In all the plots, θ is given as a percentage of the number of transactions in the database $|D|$. The frequency constraint is therefore $sup(P) \geq |D| \cdot \theta/100$ for a pattern P.

2. `https://github.com/MathieuVavrille/pattern-diversity-cp-strategy`
3. We show the results on 6 instances. All graphs can be found alongside the implementation.



(a) `hepatitis`, $\theta = 20$ (b) `chess`, $\theta = 20$ (c) `kr-vs-kp`, $\theta = 20$

(d) `T40I10D100K`, $\theta = 5$ (e) `splice1`, $\theta = 5$ (f) `pumsb`, $\theta = 30$

Figure 6.1 – Cumulative distribution function for pairwise Jaccard values for full result sets for different data

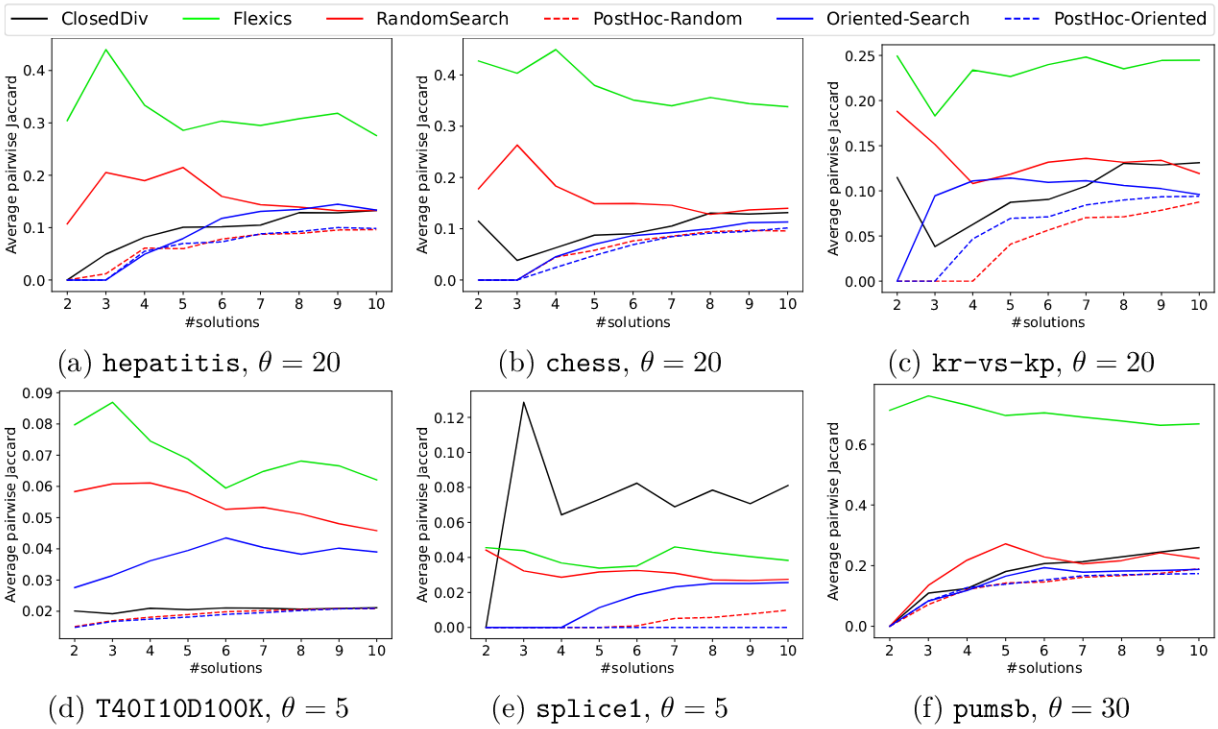
6.5.1 Global Quality of Diversification

A first way to quantify the diversity of a set of solutions is to look at the distances between all the patterns found. For each pair of patterns, we compute the Jaccard index between these two patterns, the distribution of which can be plotted. Given a set of patterns $S = \{s_1, \ldots, s_k\}$, the function we plot is the following:

$$f_{all}(\gamma) = \frac{2 \cdot \#\{(i,j) | J_{\mathcal{V}}(s_i, s_j) \leq \gamma, 1 \leq i < j \leq k\}}{k(k-1)}.$$

Given a value of Jaccard γ, $f_{all}(\gamma)$ is the proportion of pairwise Jaccard indices less than or equal to γ. The factor $\frac{2}{k(k-1)}$ is a normalisation factor (to have a result between 0 and 1). This curve, the cumulative distribution function (CDF), gives a general idea of the diversity. If the curve is high, it means that a lot of pairwise Jaccards are close to 0, which means that the solutions are diverse. Note that the order in which the solutions are found is lost in this plot.

Figure 6.1 shows the graphs on different databases. We can see that Flexics does not



(a) `hepatitis`, $\theta = 20$ (b) `chess`, $\theta = 20$ (c) `kr-vs-kp`, $\theta = 20$

(d) `T40I10D100K`, $\theta = 5$ (e) `splice1`, $\theta = 5$ (f) `pumsb`, $\theta = 30$

Figure 6.2 – Iterated average Jaccard on the different databases

return well diversified solutions in terms of Jaccard index. Since it is an approximation of diversity through a relaxed constraint, ClosedDiv does not perform well either. On the other hand, the approaches RandomSearch, OrientedSearch and their PostHoc equivalents perform better.

6.5.2 Diversity in Early Solutions

The second diversity plots focus on the first solutions found in the search process for different approaches. OrientedSearch restarts the search after each solution, to ensure a search further away from the previous solutions. At the beginning of the search, this has a strong influence on the diversity, because it can search in spaces where no solution has been found. After a number of solutions have been found, we do not expect to be able to improve the diversity much, because the solution space has already been covered. Considering that a user may not want to manually check a large number of patterns, we show the average Jaccard index between the first solutions, by iteratively increasing the number of solutions. Specifically, given a set of solutions $S = \{s_1, \ldots, s_k\}$ returned in the order $s_1, \ldots, s_k$, we plot $f_{mean}(m)$, the average Jaccard between the m first solutions

returned:

$$f_{mean}(m) = \frac{2}{m(m-1)} \cdot \sum_{1 \leq i < j \leq m} J_{\mathcal{V}}(s_i, s_j)$$

If the plot is low, it means that the average Jaccard index is close to 0 and the solutions are diverse.

Figure 6.2 shows the average pairwise Jaccard (f_{mean}) on the first 10 solutions on different databases. The PostHoc approach gives the best average Jaccard on the first solutions, because it applies a post-processing to the set of solutions. We see that OrientedSearch returns diverse early solutions and then converges to a low average Jaccard, a result in accordance with the global pairwise Jaccard plotted in the previous section. Since most of the other approaches do not consider previously found solutions, there is no good diversification between early solutions.

Remark on PostHoc's behaviour All the approaches except PostHoc return solutions in an online fashion, i.e. a user does not have to wait for all the solutions before starting to use the first ones. However, PostHoc must first compute a larger set of solutions, and then post-process them before returning the most diverse ones. Thus, a user may have to wait much longer to get the result.

6.5.3 Running Time

The running times of all the approaches on all the databases and the associated thresholds are shown in Figure 6.3. Table 6.2 also shows the running times, and the number of solutions drawn for each instance. Note that for some instances (such as `splice1` or `mushroom`) more than 10,000 solutions are generated.

First, it is clear that RandomSearch is always one of the fastest approaches. PostHoc using RandomSearch is also among the fastest approaches, often taking exactly twice as long as RandomSearch. It is only when a large number of itemsets are sampled that the second step can take a long time. For example, in the `mushroom` database, more than 10,000 solutions are returned by ClosedDiv, so the PostHoc approach searches for more than 20,000 patterns with the oracle (recall that $K = 2k$). Computing all the pairwise Jaccard indices between these patterns is already very expensive.

The running time of OrientedSearch depends strongly on the number of returned solutions. The most striking example is the only timeout of our approach, on the `splice1` database with $\theta = 2$. ClosedDiv returned 70,434 solutions, so it was too expensive for

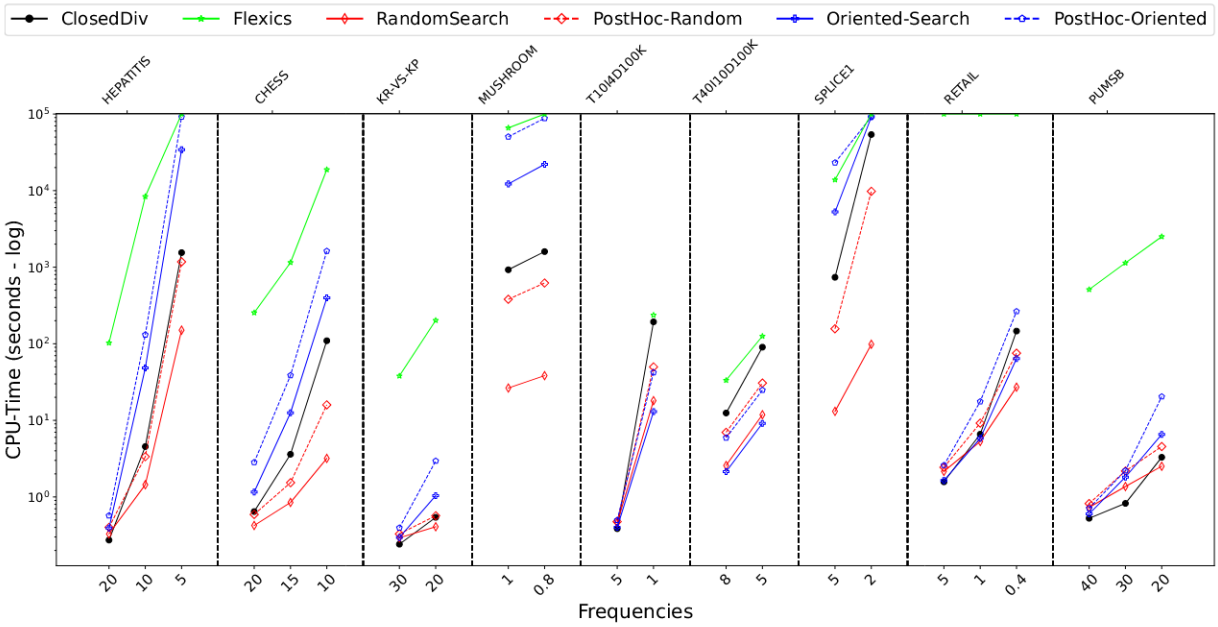


Figure 6.3 – Running times on all the databases and frequencies.

OrientedSearch to compute the Jaccard indices with all the solutions in the history at each decision step. When few solutions are returned (i.e. with a high frequency threshold), OrientedSearch is always among the fastest approaches. It performs well on sparse databases, even being the fastest approach on the `T10I4D100K` and `T40I10D100K` databases.

Flexics is the slowest approach. Uniform sampling is a strong property but it is hard to ensure, and it has a big impact on the running time.

6.5.4 Conclusion on the Experiments and Discussion

We have shown in Sections 6.5.1 and 6.5.2 that our approach yields well diversified solutions on all instances tested. For the diverse pattern mining task, only a few patterns are wanted by a user. When searching for few patterns, the running time of our approach is comparable to other state-of-the-art approaches. On these few solutions, section 6.5.2 showed that OrientedSearch almost always has the best diversity.

However, due to the computations performed when choosing which variable to branch on, searching for many solutions (in the order of thousands) leads to higher running times. We showed in section 6.5.1 that in this case RandomSearch leads to a diversity as good as that of OrientedSearch with a very low running time. The good performance of

Table 6.2 – Running times of all approaches, in seconds. The statistics of the datasets are *#items* × *#transactions*, and the density is given as a percentage. − indicates a time limit, ∗ indicates that an unexpected error occurred.

Dataset	θ	$\|S\|$	Oriented Random		PostHoc		Closed	Flexics
			Search	Search	Oriented Random		Div	
`hepatitis`	20	50	0.39	0.33	0.57	0.4	**0.27**	102.54
68 × 137	10	1340	48.3	**1.44**	131.22	3.36	4.55	8363
50%	5	33645	34263.48	**150.53**	-	1173.34	1550.15	-
`chess`	20	76	1.16	**0.42**	2.82	0.59	0.64	254.22
75 × 3196	15	294	12.55	**0.85**	38.69	1.53	3.61	1151.98
49.33%	10	2083	397.3	**3.19**	1623.87	15.95	109.62	18788
`kr-vs-kp`	30	14	0.3	0.29	0.4	0.33	**0.24**	38.02
73 × 3196	20	66	1.04	**0.41**	2.95	0.57	0.54	202.15
49.32%								
`mushroom`	1	10618	12227.21	**26.44**	50536	380.91	925.01	65909
112 × 8124	0.8	13513	21991.79	**38.36**	87838	622.12	1598.31	-
18.75%								
`splice1`	5	7920	5266.17	**13.11**	23165	156.93	737.79	13840
297 × 3190	2	70434	-	**98.46**	-	9796.2	53965	-
20.91%								
`T10I4D100K`	5	10	0.4	0.46	0.5	0.48	**0.38**	*
870 × 100000	1	359	**12.98**	18.07	42.21	49.91	193.29	236.49
1.16%								
`T40I10D100K`	8	124	**2.14**	2.57	5.93	6.98	12.47	33.36
942 × 100000	5	283	**9.14**	11.85	24.82	30.67	90.28	125.74
4.20%								
`retail`	5	11	1.64	2.16	2.59	2.43	**1.57**	-
16470 × 88162	1	104	5.82	**5.29**	17.6	9.21	6.59	-
0.06%	0.4	514	64.1	**27.1**	264.44	75.27	146.48	-
`pumsb`	40	3	0.6	0.73	0.71	0.82	**0.53**	511.29
2113 × 49046	30	13	1.81	1.37	2.21	2.18	**0.82**	1136.23
3.50%	20	41	6.55	**2.51**	20.39	4.55	3.3	2505.61

RandomSearch and OrientedSearch on either the diversity or the running time shows that the search strategies are an excellent way of using CP solvers to enforce diversity in pattern mining problems.

The PostHoc approach can be used in conjunction with either RandomSearch or OrientedSearch to further improve the diversity at the cost of a longer running time.

6.6 Related Works

We have already presented and benchmarked against ClosedDiv [109] and Flexics [107]. Other approaches have been proposed to make pattern mining more useful for exploratory purposes. Each of these solutions has its own advantages and disadvantages.

Some approaches are designed to find diverse patterns depending on some quality measure in a generic pattern mining problem. Gibbs [99] is a sampling process driven by an interestingness measure [100] updated with statistics of patterns already found. The number of iterations of the process can be increased to get a better sampling. CFTP [101] is a fast two-step random sampling procedure tailored to a limited set of itemset mining tasks, using an objective quality measure φ. Patterns' probability of being sampled is related to their score but ignores previously sampled patterns. Using $\varphi = sup$, the patterns should be sampled according to their frequency. However, these two approaches cannot deal with a strict frequency constraint. In practice, in some instances the sampled itemsets may be very small. For example, on `T40I10D100K` with $\theta = 5$, CFTP (with a parameter of 3) only samples patterns covering a single transaction, and only 2% of the patterns sampled by Gibbs satisfy the constraint we imposed on the CP (and Flexics) approaches.

Condensed representations [106] still typically leave many patterns and do not achieve diversity in the final sets. Top-k mining [120] is efficient but results in strongly related, redundant patterns. Pattern set mining [118] takes into account the relationships between the patterns, which can result in small solution sets, but just pushes the problem further down the line.

Older works on pattern set selection [105, 111, 112] have investigated alternative measures of diversity in pattern sets. Joint entropy is proposed in [111] as a quality measure to mine maximally informative k-itemsets in post-processing. Recent work, on the other hand, pushes diversity constraints into the mining process itself [98, 103]. In [103], the authors propose using MCTS and upper confidence bounds to guide the search to interesting regions in the lattice, given the already explored space. The authors of [98] propose a greedy algorithm exploiting upper bounds to iteratively extract up to k subgroup descriptions, considering sets of subgroup descriptions as disjunctions of such patterns.

6.7 Conclusion

We presented OrientedSearch, a new approach to mine diverse patterns that exploits one of the strengths of CP solvers: search strategies. We focused on the Jaccard index to measure the diversity of the solutions, but OrientedSearch is generic and other measures could be used, by simply changing the definition of the history score. We have shown experimentally that our approach can generate small sets of well-diversified solutions very efficiently. As it is based on the CP framework, users can add their own constraints to suit their needs.

Our approach takes longer to run when many solutions are requested, but in this case, RandomSearch returns patterns that are just as diverse. Despite its simplicity, RandomSearch allows the solver to find solutions very quickly. It can also be combined with a post-processing step such as PostHoc to extract only diverse patterns from those returned.

This chapter shows that search strategies are an excellent diversity approach to solve pattern mining problems. In frequent and closed itemset mining, the constraint propagates inconsistent values well, so the search strategy does not often make decisions that lead to unsatisfiable spaces. We used this to design diversity oriented search strategies.

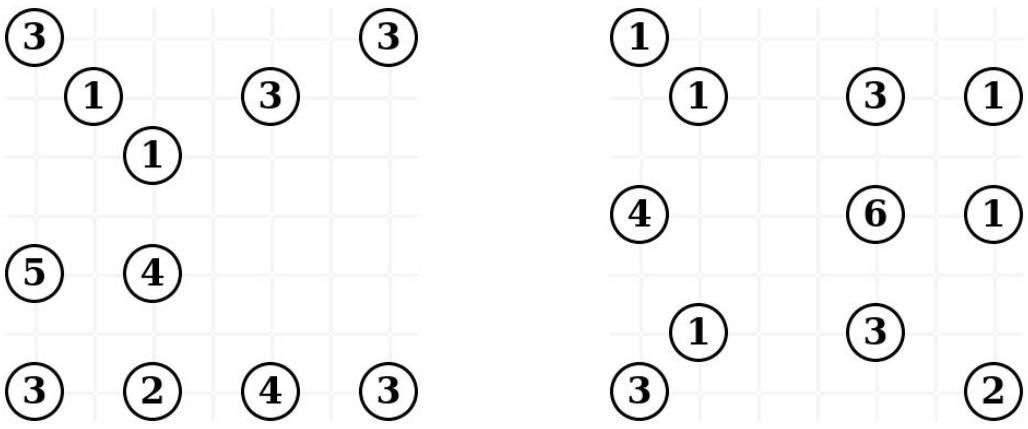


Figure 6.4 – Two *Bridges* grids, see rules in Appendix C.3.1.

Chapter 7

Feature Models

This contribution originated from discussions with Mathieu Acher on software tests generation. It continued with Erwan Meunier's internship, which I co-supervised, on a uniform sampler for feature diagrams. Following this internship, I published a research report [3], which is presented in this chapter in Section 7.6. The rest of this chapter is my work on t-wise coverage: the theoretical analysis of RandomSearch and the design of a new search strategy. A short article on the behaviour of RandomSearch was accepted and presented at the ROADEF 2023 conference. [a]

a. https://roadef2023.sciencesconf.org/434918

7.1 Introduction

Efficient testing of Product Lines is of high importance to assess quality or (in the case of Software Product Lines) the absence of bugs [138]. In highly configurable systems, this testing task is complicated by the large number of interacting features. For example, the Linux kernel contains thousands of interacting features (such as compilation options or installed libraries) [140]. Configurations (i.e. sets of features) can be tested by instantiating them on the given product line (for example by compiling the Linux kernel with specific options and libraries). These tests can be expensive (in terms running time [140], memory [130], or manpower [127]), so efficient test suites (a set of configurations) need to be generated.

One way to measure the quality of a test suite is the t-wise coverage [137]. It aims to ensure that all interactions (combinations) of up to t features are tested. But there can be $2^t\binom{n}{t}$ t-wise combinations on n features. Thus, with thousands of features, computing the t-wise combinations allowed by the product line can be prohibitive, let alone generating a minimal test suite that covers all these combinations. To overcome this issue, approaches have been developed that use approximations based on random processes such as uniform [84] or weighted [123] sampling. These approaches lose the guarantees, but the

diversity induced by the randomness allows for good experimental coverage and running times.

In this chapter, we use Constraint Programming's random search strategies to find high-coverage test suites. Search strategies are a way of making the search find solutions in different solution spaces. In particular, random search strategies do not need to compute expensive metrics (such as the number of allowed combinations) and can generate diverse (i.e. high coverage) solutions. The contributions of this chapter are as follows.

- We analyse the theoretical properties of the default random search strategy. We show that the (non-uniform) distribution of the solutions returned by this default random search strategy is well suited to the task of computing solution sets with a good t-wise coverage.
- We design an improvement to this search strategy by using information about the product line: the commonality. The commonality of a feature is the number of times it appears in all the possible configurations. We use this information to make better choices during the decisions of the search strategy, to find solutions that cover more unseen combinations.

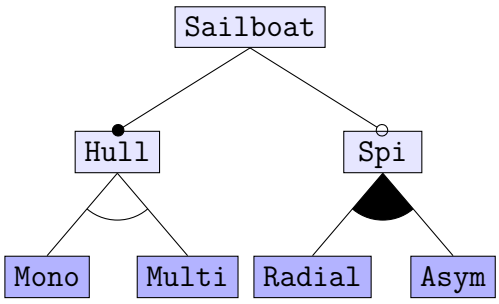
We experiment with these two search strategies and compare them to state-of-the-art sampling approaches. We show that the search strategies outperform all other approaches in the t-wise coverage and running time. Our new approach improves the default random search strategy without any running time overhead. We show that a uniform sampling is actually detrimental to the t-wise coverage.

Outline

This chapter is organised as follows. Section 7.2 defines the notions used in the rest of the chapter and Section 7.3 presents the related works. In Section 7.4 we analyse of the RandomSearch strategy. Our new strategy FrequencyDiff for finding good coverage solution sets is presented in Section 7.5. Finally, Section 7.7 presents the methodology and the results of the experiments.

7.2 Background

This section introduces the notions used in this chapter.



Cross-tree constraints:
`Multi` $\implies$ `Asym`

(a) Example of Feature Model

	1	2	3	4	5	6	7	8
`Sailboat`	∗	∗	∗	∗	∗	∗	∗	∗
`Hull`	∗	∗	∗	∗	∗	∗	∗	∗
`Mono`	∗	∗	∗	∗				
`Multi`					∗	∗	∗	∗
`Spi`	∗	∗	∗		∗	∗	∗	
`Radial`	∗		∗		∗		∗	
`Asym`	∗	∗			∗	∗		

(b) Allowed configurations. There is one column per configuration, and a ∗ means that the feature is present. Configurations 7 and 8 are allowed by the feature diagram but not the cross-tree constraint

Figure 7.1 – A feature model and its set of allowed configurations.

7.2.1 Feature Models

Feature models are a graphical and condensed representation of the products in a product line [126]. Given a fixed set of features $\mathcal{F}$, a feature model is a pair of, first, a feature diagram, which gives a hierarchical structure to the organisation of the features, and second, a conjunction of propositional formulas over $\mathcal{F}$.

Example. *Figure 7.1a shows a feature model representing sailboats. It is mandatory that a sailboat has a* ***`Hull`*** *(black dot above the* ***`Hull`*** *node) and optionally a* ***`Spi`*** *(empty dot above the* ***`Spi`*** *node), which is a special sail at the front of the boat. The* ***`Hull`*** *can be either* ***`Mono`*** *or* ***`Multi`*** *hull, but not both (represented by the arc between the two nodes). If there is a* ***`Spi`****, it can be a* ***`Radial`*** *one, an* ***`Asym`*** *(asymetric) one, or both (represented by the black arc between the two nodes). There is also a constraint that a* ***`Multi`****-hull boat must have an* ***`Asym`****metrical spi.*

Table 7.1b shows the configurations allowed by this feature model. The last two configurations (7 and 8) are allowed by the feature diagram, but not by the cross-tree constraint.

As shown in Figure 7.1a, feature models are defined using a tree structure, called a feature diagram, and cross-tree constraints. Feature diagrams define the hierarchical structure of the features in a feature model. We now formally define feature diagrams.

Definition 33 (Feature Diagram)**.** A *feature diagram* is an n-ary labelled tree, where the nodes can be of different types. A feature diagram D stores a feature $D.\texttt{feature} \in \mathcal{F}$ at its root. The children can be from:

- a `mandatory/optional` group, where the sets $D.\texttt{mand}$ and $D.\texttt{opt}$ contain the mandatory and optional children;
- an exclusive (`xor`) group, where the set $D.\texttt{xor}$ contains the children;
- an `or` group, where the set $D.\texttt{or}$ contains the children.

In addition, each feature may appear only once in the feature diagram.

This definition is a recursive definition of feature diagrams. We call D' a sub-feature diagram of D if D is an ancestor of D' or D itself. Feature diagrams restrict the allowed configurations (set of features) of the feature model.

Definition 34 (Allowed Configuration)**.** Given a feature diagram D, a configuration $C \subseteq \mathcal{F}$ is allowed iff:

- $D.\texttt{feature} \in C$
- for all D' sub-feature diagram of D, $\forall D'' \in D'.\texttt{children}, D''.\texttt{feature} \in C \Rightarrow D'.\texttt{feature} \in C$
- for all D' sub-feature diagram of F, if $D'.feature \in C$ then:
  - $\forall D'' \in D'.\texttt{mand}, D''.\texttt{feature} \in C$
  - $\exists D'' \in D'.\texttt{or}, D''.\texttt{feature} \in C$
  - $\exists! D'' \in D'.\texttt{xor}, D''.\texttt{feature} \in C$

We denote by $Sols\,(D)$ the set of allowed configurations.

Informally, all the features in $D.\texttt{mand}$ children must be taken, at least one feature in the $D.\texttt{or}$ group must be taken, and exactly one feature in the $D.\texttt{xor}$ group must be taken. When a feature is taken, its parent feature must also be taken.

To provide greater for more expressiveness when modelling feature interactions, feature diagrams are extended with propositional formulas that allow to model interactions between features that are not ancestors of each other.

Definition 35 (Feature Model)**.** A *Feature Model* M is a pair $\langle D, \psi \rangle$ where D is a feature diagram and ψ is a Boolean formula where the variables are features contained in $\mathcal{F}$. A configuration is allowed by M if it is allowed by D and satisfies the Boolean formula ψ. We note $Sols\,(M)$ the set of allowed configurations.

The propositional formulas allow for more diverse constraints, but also make the problem much harder, as simply finding a configuration is NP-complete. Some definitions restrict the cross-tree constraints in ψ to the implication or exclusion of features [126], but this has been shown to reduce the expressiveness [136].

Feature models can be translated into CNF formulas, where an instantiation corresponds to a unique configuration [125]. The model defines a variable X_f for each feature f. However, the conversion of propositional formulas can lead to an exponential number of clauses [29]. To prevent this exponential explosion, CP was used in [134, 135] (for extensions of feature models to integer variables and global constraints).

Example. *The* `Sailboat` *feature model defined in Figure 7.1a can be modelled by a CSP with Boolean variables* X_F *for* `F` *a feature, and the constraints*

$$
\begin{aligned}
X_{Sailboat} &= 1 && \textit{(root feature)}\\
X_{Hull} &= X_{Sailboat} && \textit{(mandatory child)}\\
X_{Mono} + X_{Multi} &= X_{Hull} && \textit{(exclusive children)}\\
X_{Spi} &\Rightarrow X_{Sailboat} && \textit{(child implies parent)}\\
X_{Radial} &\Rightarrow X_{Spi} && \textit{(child implies parent)}\\
X_{Asym} &\Rightarrow X_{Spi} && \textit{(child implies parent)}\\
X_{Radial} + X_{Asym} &\geq X_{Spi} && \textit{(or children)}\\
X_{Multi} &\Rightarrow X_{Asym} && \textit{(cross-tree constraint)}.
\end{aligned}
$$

7.2.2 t-wise Coverage

In this chapter, the metric used to assess the quality of a test suite is the t-wise coverage. This metric focuses on testing interactions of t features, through means of combinations.

Definition 36 (t-wise Combination)**.** A *t-wise combination* is a mapping $\sigma : \mathcal{F}' \to \{0, 1\}$ with $\mathcal{F}' \subseteq \mathcal{F}$ and $|\mathcal{F}'| = t$.

A configuration C covers a t-wise combination σ iff $\forall f \in \mathcal{F}', \sigma(f) = 1 \Leftrightarrow f \in C$. In this case we say that σ is included in the configuration, and write $\sigma \subset C$. We note $Comb_t(C)$ all the t-wise combinations covered by a configuration C (if there is no ambiguity about the value of t, we omit it in $Comb_t$). By extension, given a test suite $\mathcal{S}$, $Comb(\mathcal{S})$ is the set of combinations covered by at least one configuration of $\mathcal{S}$ (i.e. $Comb(\mathcal{S}) = \bigcup_{C \in \mathcal{S}} Comb(C)$).

Given a feature model M, a combination is said to be possible if there is a combination in $Sols\,(M)$ that covers it. For simplicity we note $Comb(M) = Comb(Sols\,(M))$ all the combinations covered by at least one configuration allowed by M. The coverage of a test

suite is the fraction of possible combinations that are covered, i.e.

$$Cov(\mathcal{S}) = \frac{|Comb(\mathcal{S})|}{|Comb(M)|}\,.$$

As t grows, the number of t-wise combinations grows exponentially. Indeed, the number of possible combinations can be as large as $\binom{|\mathcal{F}|}{t}2^t$. The number of combinations covered by a single configuration also grows exponentially as t grows, and is equal to $\binom{|\mathcal{F}|}{t}$.

Ideally, all the interactions of features are tested, so that all the $|\mathcal{F}|$-wise combinations are covered, but in practice this is impossible due to the exponential growth of the number of combinations. A study by the NIST [138] states that most of the faults/bugs in software come from up to 6-wise combinations. This greatly reduces the number of combinations to test, but for large feature models it would still not be reasonable to try to enumerate all the possible 6-wise combinations.

7.2.3 Links between Commonalities and Uniform Sampling

This section recalls the properties of t-wise coverage of uniform samplers stated in [141]. A sampler is a random selection process whose result is not deterministic. On a feature model M, a sampler $\mathcal{U}$ generates a random allowed configuration, i.e. $\mathcal{U}(M)$ is a random variable taking values in the set $Sols\,(M)$. We recall the definition of a uniform sampler, adapted to feature models.

Definition 15' (Uniform Sampler)**.** Let M be a feature model. A function $\mathcal{U}$ is a uniform sampler on M iff

$$\forall C \in Sols\,(M)\,, \mathbb{P}\,(\mathcal{U}(M) = C) = \frac{1}{|Sols\,(M)|}\,.$$

Applied to configurations of feature models, uniform sampling can generate a test suite. It has already been used on feature models in SMARCH [84] and extended to weighted sampling in BAITAL [123]. There is no guarantee of t-wise coverage, but there is no need to compute the exponential set of all t-wise combinations: the diversity is provided by randomness.

When using random algorithms, since there are fewer guarantees, it may be useful to have information about the average behaviour. This average behaviour of uniform samplers on the t-wise coverage was studied in [141]. The authors found that the t-wise coverage depends on the commonalities of the combinations, defined as follows.

Definition 37 (Commonality)**.** The *commonality* of a combination σ in a feature model M, noted φ_σ, is its frequency of occurrence in the set of allowed configurations, i.e.

$$\varphi_\sigma = \frac{|\{C \in Sols(M) \mid \sigma \subset C\}|}{|Sols(M)|}$$

Commonalities provides important information about the feature model. It allows to know which combinations are more common in the set of allowed configurations. A user may want to design a test suite that covers the frequent combinations, as these may be the most used, or conversely, a user could focus on low commonalities to test combinations that may have been missed by other tests.

The problem of computing the commonality of a configuration is hard, because it requires calls to a #-SAT solver. For example, the strategy 3 of baital [123] makes $|\mathcal{F}|+1$ calls to a #-SAT solver to compute all the commonalities of features. For large feature models this can be prohibitively expensive. If the cross-tree constraints are dropped (only the feature diagram is considered), it is possible to compute the commonalities for all the features (1-wise combinations) in linear time [129, 3]. This quickly gives an approximation of the commonality of the features.

Uniform samplers guarantee that all solutions have the same probability of being returned. For the t-wise coverage, however, we are interested in the probability that a t-wise combination is returned by the sampler. The following proposition states that this probability is equal to the commonality of the combination.

Proposition 1 ([141])**.** *Let M be a feature model, $\mathcal{U}$ be a uniform sampler (i.e. $\forall C \in Sols(M), \mathbb{P}(\mathcal{U}(M) = C) = 1/|Sols(M)|$), and σ be a combination, then*

$$\mathbb{P}(\sigma \subset \mathcal{U}(M)) = \varphi_\sigma .$$

Proof. We use the definition of the probability of a random event (positive cases divided by total cases), and the definition of the commonality

$$\begin{aligned}\mathbb{P}(\sigma \subset \mathcal{U}(M)) &= \frac{\#\text{positive cases}}{\#\text{total cases}} \\ &= \frac{|\{C \in Sols(M) \mid \sigma \subset C\}|}{|Sols(M)|} \\ &= \varphi_\sigma\end{aligned}$$

□

The probability of having a given combination in the solution returned by a uniform sampler is equal to the commonality of the combination. This means that if there are features with very low commonality, a sampler may never return a solution containing them in a reasonable number of samples. For example, the authors in [141] remarked that on their experiments, 38.8% of the features (1-wise combinations) have a commonality $\varphi_\sigma < 0.0001$, so it is very unlikely that a uniform sampler will produce a solution containing these features.

Notation. *Given a sampler $\mathcal{A}$ (uniform or not), a feature model M, and a t-wise combination σ, we note $p_\sigma^{\mathcal{A}}(M) = \mathbb{P}(\sigma \subset \mathcal{A}(M))$ the probability that the sampler $\mathcal{A}$ returns a solution that covers the combination σ. In the following, when there is no ambiguity in the feature model, we simply write $p_\sigma^{\mathcal{A}}$. In this chapter, we consider two types of samplers. For a uniform sampler, noted $\mathcal{U}$, proposition 1 states that $p_\sigma^{\mathcal{U}} = \varphi_\sigma$. For a sampler based on the RandomSearch search strategy, the probability $p_\sigma^{\mathcal{R}}$ is unknown, we analyse it in Section 7.4.*

For the task of t-wise coverage, if many of combinations are unlikely to be found, the test suite would not have a good t-wise coverage. Let $\mathcal{S}_n$ be a test suite generated by calling a sampler n times independently on a feature model M. $\mathcal{S}_n$ is a random variable taking values in $Sols(M)^n$. The t-wise coverage of $\mathcal{S}_n$, $Cov(\mathcal{S}_n)$, is also a random variable taking values in $[0, 1]$. To evaluate the behaviour of a sampler in terms of t-wise coverage, we are interested in the expected value of $Cov(\mathcal{S}_n)$, i.e. $\mathbb{E}(Cov(\mathcal{S}_n))$. A formula for this expected t-wise coverage is given in [141] in the case of uniform samplers. We recall and prove it here in the general case (for any sampler).

Proposition 2 ([141])**.** *Let M be a feature model. Let $p_\sigma^{\mathcal{A}}$ be the probability that a sampler $\mathcal{A}$ (uniform or not) returns a solution containing the combination σ. Let $\mathcal{S}_n$ be a set of n configurations of a feature model M, generated by a such a sampler (by calling it n times independently). $\mathcal{S}_n$ is a random variable, and so is the set of t-wise combinations covered $Cov(\mathcal{S}_n)$. The expected value of $Cov(\mathcal{S}_n)$ is*

$$\mathbb{E}(Cov(\mathcal{S}_n)) = \frac{1}{|Comb(M)|} \cdot \sum_{\sigma \in Comb(M)} (1 - (1 - p_\sigma^{\mathcal{A}})^n) \,.$$

Proof. Given a combination σ, we define the random variable Z_σ to be equal to 1 iff σ is covered by a solution in $\mathcal{S}_n$ (and 0 otherwise). Then, $|Comb(\mathcal{S}_n)| = \sum_{\sigma \in Comb(M)} Z_\sigma$. Due

to the linearity of the expected value,

$$\mathbb{E}(Cov(\mathcal{S}_n)) = \frac{1}{|Comb(M)|} \cdot \sum_{\sigma \in Comb(M)} \mathbb{E}(Z_\sigma)\,.$$

Z_σ follows a Bernoulli distribution (it has only 2 possibilities), hence $\mathbb{E}(Z_\sigma) = \mathbb{P}(Z_\sigma = 1) = 1 - \mathbb{P}(Z_\sigma = 0)$. The probability that σ is not covered by any of the configurations of $\mathcal{S}_n = \{C_1, \ldots, C_n\}$ is

$$\begin{aligned}
\mathbb{E}(Z_\sigma) &= 1 - \mathbb{P}(Z_\sigma = 0) \\
&= 1 - \prod_{i=1}^{n} \mathbb{P}(\sigma \not\subset C_i) && \text{(independence of solutions)} \\
&= 1 - (1 - p_\sigma^{\mathcal{A}})^n
\end{aligned}$$

□

This proposition confirms the intuition that the expected t-wise coverage depends on the probability of sampling each combination. If there are combinations with a low sampling probability of being sampled, the expected coverage will increase more slowly (when the number of solution increases) than if all the combinations had a high sampling probability of being sampled. In Section 7.4 we prove a lower bound on the sampling probability $p_\sigma^{\mathcal{R}}$ when using RandomSearch.

7.3 Related Works

7.3.1 Dedicated Approaches

t-wise coverage is a well-studied problem in feature models. Most approaches either require access to the set of possible combinations (by making $\binom{|\mathcal{F}|}{t}2^t$ calls to a SAT solver), or will iteratively generate this exponential set. These approaches often have the guarantee that all combinations are covered, at the cost of an expensive generation of combinations. AETG [128] is one of the first algorithms for t-wise coverage. The authors propose a way to select variables to set (or forbid) in the searched configuration based on the combinations that have not yet been covered.

ICPL [133] is based on the fact that a t-wise covering test suite is a good starting point for generating a $t+1$-wise covering test suite. This remark also allows to speed up

the generation of possible $t+1$-wise combinations, since some of them were detected as impossible by the t-wise test suite.

In IncLing [122] the authors propose several improvements (such as the detection of dead or core features and a feature ranking heuristic) in an incremental sampling. In [143], the authors propose to use the advances in SAT solvers to detect impossible combinations more efficiently (instead of making a SAT call to verify each combination). By using unsatisfiability cores returned by SAT solvers on unsatisfiability, they can reduce the set of potential combinations. This greatly reduces the number of SAT calls required to find a covering test suite.

7.3.2 Sampling Based Approaches

All these approaches have the guarantee of generating a covering test suite, but at the cost of having to generate the set of possible combinations, which can be prohibitive for large feature models or t. A well-known approach to search for diverse configurations is to use randomness, and for example a uniform sampler. Recent advances in uniform SAT samplers such as Smarch [84] can efficiently generate configurations of large feature models. This approach does not guarantee coverage and it has been shown experimentally in [141] that up to 10^{14} configurations (almost the enumeration of all solutions) need to be generated to achieve 100% coverage on some feature models.

Baital [123] corrects this issue by using the weighted sampler WAPS [80]. This sampler compiles the cnf formula representing a feature model into a d-DNNF representation. This representation is then annotated with weights, and a weighted sampling can be performed very efficiently. Baital will perform r rounds, each rounds drawing s samples will be drawn from a given distribution. At the start of each round, the distribution is modified by changing the annotation of the d-DNNF representation depending on the solutions found. If a feature has only been sampled a few times, its weight is increased to increase the probability of sampling a configuration containing it. Each annotation phase is costly, but it helps to find new combinations. We compare our approach to Baital in the experimental section 7.7.

CMSGen [78] is a recent SAT sampler that has been shown to generate test suites with higher coverage than Baital. CMSGen modifies a SAT sampler to use the RandomSearch strategy (picking a random uninstantiated variable, and a random value between 0 and 1). This sampler is therefore non-uniform (this can even be shown on a problem with two variables x and y, and the clause $x \vee y$). In Section 7.4 we analyse the

behaviour of RandomSearch (and thus CMSGen) on the task of test suite generation for t-wise coverage. We show why RandomSearch is well suited for this task, and at the same time explain the reasons for the great results of CMSGen. In the experiments we compare to RandomSearch, i.e. the CP version of CMSGen.

7.3.3 In Constraint Programming

Constraint Programming provides a variety of modelling and solving tools (such as search strategies and global constraints). In Pacogen [131, 132], constraint programming is used to find the smallest test suite that ensures full pairwise coverage. The authors propose a data structure (in a matrix) to store the full test suite to be generated, and a global constraint to ensure that the pairwise combinations are covered. As other approaches, the combinations need to be enumerated, so this approach would not scale to larger t and large feature models (with thousands of features).

Recently, advances done in hashing-based SAT samplers (see the Unigen [69] line of work) have been extended to CP. By adding random hashing constraints to the model, the solution set is randomly cut inso small cells. In Chapter 5 we presented TableSampling [2], a sampling algorithm that uses `table` constraints as hashing tables. A table constraint is a constraint given in extension, i.e. for a given subset of variables, all the allowed instantiations are given to the constraint. In a t-wise framework, this can be seen as allowing or disallowing some t-wise combinations on a given set of variables. We evaluate TableSampling in Section 7.7 to see if such hashing constraints based on table constraints lead to a good t-wise coverage.

7.4 RandomSearch's Behaviour

As shown in Proposition 2, the t-wise coverage of a sampler $\mathcal{A}$ is related to its probability $p_\sigma^{\mathcal{A}}$ of sampling a given combination σ. In the case of a uniform sampler $\mathcal{U}$, Proposition 1 states that $p_\sigma^{\mathcal{U}} = \varphi_\sigma$. For a sampler $\mathcal{R}$ using the RandomSearch strategy, this probability (noted $p_\sigma^{\mathcal{R}}$) is unknown: we study it here.

7.4.1 Example for a Single Feature

The main difference between uniform sampling and RandomSearch is that uniform sampling focuses on configurations, whereas RandomSearch focuses on features. At the

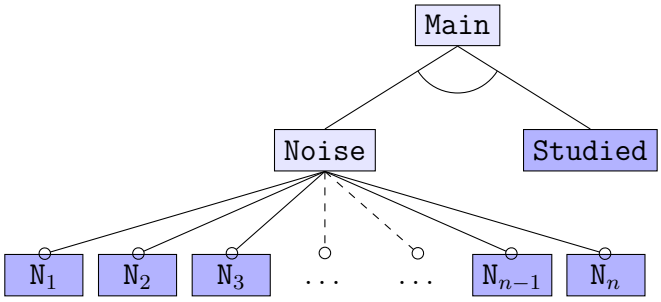


Figure 7.2 – Example of noisy Feature Model

decision level, all the features have the same probability of being picked (or removed). The toy feature model in Figure 7.2 illustrates this behaviour. At the root node `Main`, this feature model has two exclusive children `Noise` and `Studied`. We are interested here in the `Studied` feature, which is only contained in the configuration $\{\texttt{Main}, \texttt{Studied}\}$. However, there are many other configurations the `Noise` feature is selected. This feature has n optional children, so there are 2^n possible configurations.

On the one hand, it is very unlikely that a uniform sampler will generate the solution containing the feature `Studied`, because it is flooded among other configuration. The exact probability of sampling feature `Studied` is $p^{\mathcal{U}}_{\texttt{Studied}} = \varphi_{\texttt{Studied}} = 1/(2^n + 1)$ (there is only one configuration containing `Studied`, but 2^n noisy configurations).

On the other hand, RandomSearch is much more likely to sample the feature `Studied`. The CSP representation of a feature model contains one variable per feature. We make the CSP explicit for the example 7.2.

Example. *To represent the feature model given in Figure 7.2 as a CSP, one variable X_F is created for each feature ($\mathcal{F} = \{\texttt{Main}, \texttt{Studied}, \texttt{Noise}, \texttt{N}_1, \ldots, \texttt{N}_n\}$). All the variables have a Boolean domain. The constraints are the following:*

- *The root node must be selected:*

$$X_{\texttt{Main}} = 1\,.$$

- *A* **`xor`** *node (the* **`Main`** *node) states that if the parent is selected, only one child is selected:*

$$X_{\texttt{Studied}} + X_{\texttt{Noise}} = X_{\texttt{Main}}\,.$$

- *If an optional child is selected, then the parent must be selected:*

$$\forall i \in \{1, \dots, n\}, X_{N_i} \Rightarrow X_{Noise}\,.$$

When all the variables are instantiated, the combination can be retrieved by keeping all the features whose variables take the value 1, i.e. if S is the solution to the CSP, the associated configuration is $C = \{F \in \mathcal{F} \mid S(X_F) = 1\}$.

At the start of the search, $X_{\texttt{Main}}$ is propagated to 1, and no more propagation can be done. A decision is then computed. There are $n + 2$ uninstantiated variables: $X_{\texttt{Studied}}$, $X_{\texttt{Noise}}, X_{\texttt{N}_1}, \dots, X_{\texttt{N}_n}$. RandomSearch chooses a variable uniformly, so it has a probability of $1/(n + 2)$ to choose $X_{\texttt{Studied}}$. A value is chosen randomly, so it has a probability $1/2$ of being 1. In this case, the decision $X_{\texttt{Studied}} = 1$ is pushed to the search, and the solution will contain the feature `Studied`. Overall, there is at least a probability $\frac{1}{2(n+2)}$ that a solver using RandomSearch will produce a solution containing `Studied`.

This toy example shows the advantage of RandomSearch for the task of t-wise coverage. Choosing each variable with the same probability ensures that each variable has a non-negligible chance of being taken.

7.4.2 Generalising to Multiple Features

Having understood the behaviour for a single feature, we can now generalise the reasoning for multiple features, hence t-wise combinations. In this section, we first give and prove a loose bound. This proof gives the intuition for the behaviour of RandomSearch on t-wise combinations. We then give a tighter bound, and prove it in Appendix A.1. This proof is a refinement of the one presented here, but is more technical and it does not give more insight into the behaviour.

Proposition 3. *On a feature model with n features, the probability $p_\sigma^{\mathcal{R}}$ of sampling an allowed t-wise combination σ can be lower bounded by*

$$p_\sigma^{\mathcal{R}} \geq \frac{1}{2^t \binom{n}{t}}$$

Proof. We restrict our analysis to the first t decisions made during the search. To be sure that the sampled solution contains σ, then these first decisions must be made on the features of σ. In the first decision, there is a $\frac{t}{n}$ chance of choosing one of the variables of

σ. On the second decision, there may be only $m \leq n-1$ uninstantiated variables left. There is a $\frac{t-1}{m} \geq \frac{t-1}{n-1}$ chance to choose a second variable of σ during the second decision. Continuing the reasoning, the probability of choosing all the variables of σ during the t first decisions is greater than

$$\frac{t}{n} \cdot \frac{t-1}{n-1} \cdots \frac{1}{n-t+1} = \frac{1}{\binom{n}{t}}.$$

To get exactly the combination σ, the chosen values for every variable $X_{\mathtt{F}}$ must be the one in the combination, i.e. $\sigma(X_{\mathtt{F}})$. For each decision there are two two choices with same probability, hence the factor $\frac{1}{2^t}$. □

We are interested in the case where t is small (less than 7 as remarked in [138]) and n is large. In this setting, the proposition can be refined by the following theorem, which gives a lower bound as a convergence result.

Theorem 5. *Given a feature model with n features, and σ an allowed t-wise combination, there is a sequence u_n^t such that*

$$p_\sigma^{\mathcal{R}} \geq u_n^t$$

and

$$u_n^t \underset{n\to\infty}{\sim} \frac{1}{\binom{n}{t}}.$$

Proof. In Appendix A.1. □

Informally, $p_\sigma^{\mathcal{R}}$ can be approximately lower bounded by $1/\binom{n}{t}$. Compared to the previous proposition, the factor $1/2^t$ has been dropped.

If t is fixed to a small value, this lower bound is polynomial in n, the number of features in the feature model. On the other hand, for uniform sampling, the sampling probability can only be lower bounded by $1/2^n$, as seen in the previous example in Figure 7.2. We recall that Proposition 2 states that the lower the sampling probabilities, the worse the expected t-wise coverage. The polynomial lower bound for the sampling probability using RandomSearch is an argument for the fact that it is a sampling method that generates test suites with high t-wise coverage. We prove this fact experimentally in Section 7.7.

1 **Function** FrequencyDiff($\mathcal{P}, \mathcal{F}, \varphi, \varphi^{obs}$)
  Data: A CSP $\mathcal{P} = \langle (X_1, \ldots, X_n), \mathcal{D}, \mathcal{C} \rangle$, a set of solutions $\mathcal{S}$, a list of features (associated to variables) $\mathcal{F} = \{F_1, \ldots, F_n\}$, a mapping φ giving the commonality of every feature, and a mapping φ^{obs} giving the observed frequency of every feature in the previous solutions.
  Result: A decision.
2 $W \leftarrow$ array of size n (indexed from 1) initialized at 0;
3 **for** $i = 1$ *to* n **do**
4 **if** $\mathcal{D}(X_i) = \{0, 1\}$ **then**
5 $W[i] \leftarrow \left|\varphi^{obs}_{F_i} - \varphi_{F_i}\right|$;
6 $varId \leftarrow$ PickWeightedRandom(W);
7 **if** $\varphi^{obs}_{F_{varId}} > \varphi_{F_{varId}}$ **then**
8 $chosenValue \leftarrow 0$;
9 **else**
10 $chosenValue \leftarrow 1$;
11 **if** $Random() > \frac{1+W[varId]}{2}$ **then**
12 $chosenValue \leftarrow 1 - chosenValue$;
13 **return** Decision($X_{varId} = chosenValue$);

Algorithm 7.1: Computation of the decision of FrequencyDiff

7.5 Frequency Difference Search Strategy

The previous section showed that RandomSearch is a good starting point for a search strategy to generate a good coverage test suite. However it lacks insight into the solutions found previously. It can only avoid returning a solution that has already been found.

7.5.1 Presentation of the Algorithm

We now present the search strategy we have designed to generate a test suite with high t-wise coverage. It is an improvement over RandomSearch which uses the knowledge of the solutions already returned. We also use the commonalities to guide the search. We call this new search strategy FrequencyDiff because it uses the difference between the observed frequency of features, and the commonalities. In this section, we suppose that we have access to the commonalities of all features F in φ_F. In practice, we use an approximation of the commonalities. The experiments in Section 7.7 show that an approximation of the commonalities is sufficient to outperform other approaches.

Our approach is a search strategy, i.e. the choice of an uninstantiated variable, and a value in its domain to branch on during the solving process. This search strategy guides the search towards an interesting solution. When a solution is found, the search is restarted. It is presented in Algorithm 7.1. The search strategy has access to the model $\mathcal{P}$, the features $\mathcal{F}$ (such that X_i is the variable associated with the feature F_i), the commonalities for each feature φ, and the observed frequency of each feature φ^{obs}. Given a set of previously found solutions $\mathcal{S}$, the observed frequency of a feature F is $\varphi_F^{obs} = \frac{|\{C \in \mathcal{S} | F \in C\}|}{|\mathcal{S}|}$. These observed frequencies can be updated in time $\mathcal{O}(|\mathcal{F}|)$ when a solution is found.

The strategy first computes the absolute difference between the observed and theoretical frequencies and stores it in an array of weights W. These absolute differences are a quantification of how some features are underrepresented by the current set of solutions. The goal of FrequencyDiff is to correct this underrepresentation by increasing the random weights of such features.

The next step of the algorithm, line 6, is to choose the decision variable. An index is chosen according to the weights in W (i.e., $\mathbb{P}(varId = i) = W[i] / \sum_j W[j]$) [192]. This weighted choice will favour the features that have an observed frequency far from their commonalities. We want to point out that this also applies to the absence of features.

Example. *Using the example feature model in Figure 7.1a, and assuming that the first configuration returned is the fifth one (in Table 7.1b), containing the* `Multi` *feature. The* `Multi` *feature has a commonality of* $\frac{1}{3}$ *(because it only appears in configurations 5 and 6), but it has an observed frequency of* 1*, so its weight will be* $\frac{2}{3}$*. This weight is high, which increases the chance of returning configurations* not *containing* `Multi`*.*

The next and final step is to choose the value associated with the variable in the decision. This choice is made in two steps. In a first step, the chosen value is determined depending on the comparison between the observed frequency and the commonality. If the observed frequency is higher than the commonality, then we choose the value 0, to exclude the feature from the constructed configuration. Otherwise, a value of 1 is chosen to include the feature in the configuration. Then, in a second step (line 11), a random swap is performed. The probability that the value is swapped is proportional to the difference between the observed frequency and the commonality (stored in the array W). If this difference is close to 1 (i.e. there is a big under (or over)-representation of the feature), then it is unlikely that the chosen value will be swapped.

To summarise the search strategy: weights are computed with the difference between the observed frequency and the commonality, a variable is drawn according to these

weights, the value is chosen to bring the observed frequency closer to the commonality, and this value is swapped with a small probability.

7.5.2 Design Choices

We have chosen to randomise the choice of the value (line 11) to prevent the search from getting stuck in an unsatisfiable subspace. It is possible that some features are always present or absent (called *core* or *dead* features) due to some constraints. It is not trivial to check whether a feature is a core or a dead feature (unless it is trivially a *core* feature from the feature diagram, such as a mandatory child of the root node). We have chosen not to perform this check to avoid a pre-processing step.

If a core or dead feature is still uninstantiated, our search strategy may choose it as the decision variable. Then the value chosen would be 0 in the case of a core feature in line 8, or 1 in the case of a dead feature in line 10. When the solver makes this decision, it enters an unsatisfiable sub-space. Many computations and backtracks may be necessary to leave this sub-space.

To avoid failing in that case, we used two techniques. We first we added randomisation to the value. This way there is a small chance of not entering the unsatisfiable sub-space. If the search did enter such a sub-space, we use restarts monitoring the number of fails during the search. If there are too many fails (i.e. backtracks), the search is restarted from scratch. Randomising the value allows to avoid making the same bad decisions again and again when restarting.

7.6 Efficient Computations on Feature Diagrams

Computing the commonality on feature models requires to make calls to a #SAT solver. This is due to the presence of the cross-tree constraints. However, if we consider only the feature diagram, computing the commonalities becomes much easier (because it has a tree-like structure).

7.6.1 Variation Degree

The variation degree is the number of configurations allowed by a feature model. It can be computed recursively thanks to the following formulas.

Theorem 6 (Variation Degree of Feature Diagrams [139])**.** *Let D be a feature diagram. Then*

- *If $D.\texttt{children} = \emptyset$, then $|Sols(D)| = 1$.*
- *If $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$,*

$$|Sols(D)| = \prod_{D' \in D.\texttt{mand}} |Sols(D')| \times \prod_{D' \in D.\texttt{opt}} |Sols(D')| + 1\,.$$

- *If $D.\texttt{xor} \neq \emptyset$,*

$$|Sols(D)| = \sum_{D' \in D.\texttt{xor}} |Sols(D')|\,.$$

- *If $D.\texttt{or} \neq \emptyset$,*

$$|Sols(D)| = \left(\prod_{D' \in D.\texttt{or}} |Sols(D')| + 1 \right) - 1\,.$$

Proof. In Appendix A.2 □

This theorem naturally leads to a procedure to recursively compute the variation degree can naturally be derived. This procedure has a complexity linear in the number of features, and also computes the variation degree of each sub-feature diagram. All these results can be memoized for later access in constant time.

Example. *We show the computation of the variation degree on the example of Figure 7.1. We note by D_f the feature diagram rooted in feature f.*

- *The variation degree of all leaves is 1 (singleton product), so for all D in $\{D_{\texttt{Mono}}, D_{\texttt{Multi}}, D_{\texttt{Radial}}, D_{\texttt{Asym}}\}$, $|Sols(D)| = 1$.*
- *$D_{\texttt{Hull}}$ is an* **xor** *node, so the variation degrees of the children are added: $|Sols(D_{\texttt{Hull}})| = 2$.*
- *$D_{\texttt{Spi}}$ is an or node, so $|Sols(D_{\texttt{Spi}})| = (|Sols(D_{\texttt{Radial}})|+1)\cdot(|Sols(D_{\texttt{Asym}})|+1)-1 = 3$.*
- *The root node, $D_{\texttt{Sailboat}}$, is a mandatory/optional node. Using the formula we have $|Sols(D_{\texttt{Sailboat}})| = |Sols(D_{\texttt{Hull}})| \cdot (|Sols(D_{\texttt{Spi}})| + 1) = 8$.*

7.6.2 Feature Commonality

The variation degree is an important value to know before trying to enumerate all the solutions. However, it does not provide information about specific features: depending on

the structure of the feature model, some features may appear more often than others in the set of allowed configurations. The commonalities of 1-wise combinations (i.e. features) give an insight into the presence of features in the allowed configurations.

As for the variation degree, the commonality of a feature can be computed with a recursive function thanks to the following formulas.

Theorem 7 (Commonalities on Feature Diagrams [129])**.** *Let f be a feature and D be a feature diagram. We note $\phi_f(D) = |\{C \in Sols(D) \,|\, f \in C\}|$ the number of occurrences of a feature in the set of allowed configurations. Then*

$$\phi_f(D) = \begin{cases} |Sols(D)| & \text{if } D.\texttt{feature} = f \\ \frac{|Sols(D)|}{|Sols(D')|} \cdot \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{mand} \\ \frac{|Sols(D)|}{|Sols(D')|+1} \cdot \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{opt or } D' \in D.\texttt{or} \\ \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{xor} \end{cases}$$

The commonality of f in D can then be computed with $\varphi_f D = \frac{\phi_f(D)}{|Sols(D)|}$.

Proof. In Appendix A.2 □

From this theorem, we naturally derive a recursive computation method for the number of occurrences of a single feature. The computation of the commonality of all features of D can also be done in a single traversal of D, leading to a complexity that is linear in the number of features. The algorithm for computing the commonality for each feature is given in Algorithm 7.2.

The commonality of each feature in the feature diagram is a rough approximation of the commonality in the whole feature model (including the propositional formulas). However the problem of computing the commonality (or even the variation degree) is much harder in the general case, and requires calls to a #-SAT solver. For example the strategy 3 of Baital [123] makes $|\mathcal{F}| + 1$ calls to a #-SAT solver to compute all the commonalities. For large feature models this may be prohibitive.

7.6.3 Uniform Sampler

In addition to providing information on the software product line, the variation degree can also be used to perform uniform sampling. The tree-like structure of the feature diagrams can be used to design a recursive sampler.

```
1  Function OCCURRENCES(R)
       Data: A feature diagram R
       Result: A mapping σ : F → ℕ from features to their number of occurrences
2      σ ← {}
3      OCCURRENCESREC(R, 1, σ)
4      return σ
5  Procedure OCCURRENCESREC(R, κ, σ)
       Data: A feature diagram R, an integer κ for the recursive factor and a
             mapping σ.
       Result: Nothing is returned, but σ is filled with the features present in R.
6      σ[R.feature] ← κ · |Sols(R)|
7      for R' ∈ R.mand do
8          OCCURRENCESREC(R', (|Sols(R)| / |Sols(R')|) κ, σ)
9      for R' ∈ R.opt ∪ R.or do
10         OCCURRENCESREC(R', (|Sols(R)| / (|Sols(R')|+1)) κ, σ)
11     for R' ∈ R.xor do
12         OCCURRENCESREC(R', κ, σ)
```

Algorithm 7.2: Computation of the number of occurrences of every feature in the set of allowed configurations.

Proposition 4 (Uniform Sampler on Feature Diagrams)**.** *Given a feature diagram D, the following recursively defined algorithm $\mathcal{U}^{FD}$ is a uniform sampler.*

- *If $D.\texttt{children} = \emptyset$, then $\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\}$*
- *If $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$,*

$$\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\} \cup \bigcup_{D' \in D.\texttt{mand}} \mathcal{U}^{FD}(D') \cup \bigcup_{D' \in D.\texttt{opt}} \begin{cases} \emptyset & \text{with probability } \frac{1}{|Sols(D')|+1} \\ \mathcal{U}^{FD}(D') & \text{otherwise} \end{cases}$$

- *If $D'.\texttt{xor} \neq \emptyset$, choose $D' \in D.\texttt{xor}$ with probability $\frac{|Sols(D')|}{|Sols(D)|}$, then*

$$\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\} \cup \mathcal{U}^{FD}(D')$$

- *If* $D.\texttt{or} \neq \emptyset$*, we define*

$$C = \bigcup_{D' \in D.\texttt{or}} \begin{cases} \emptyset & \text{with probability } \frac{1}{|Sols(D')|+1} \\ \mathcal{U}^{FD}(D') & \text{otherwise} \end{cases}$$

and

$$\mathcal{U}^{FD}(D) = \begin{cases} \{D.\texttt{feature}\} \cup C & \text{if } C \neq \emptyset \\ \mathcal{U}^{FD}(D) & \text{otherwise} \end{cases}$$

Proof. In Appendix A.2 □

Remark. *In the definition of the uniform sampler* $\mathcal{U}^{FD}$*, in the* $D.\texttt{or} \neq \emptyset$ *case, there is a recursive call with the same feature diagram. This is the case where* $C = \emptyset$ *which is forbidden (at least one child of* $D.\texttt{or}$ *has to be taken). In this case, we simply generate a new configuration* C *by recursively calling* $\mathcal{U}^{FD}(D)$*. The probability of* C *being empty (i.e. the probability of calling* $\mathcal{U}^{FD}(D)$ *again) is* $\frac{1}{|Sols(D)|+1}$*, so it is very unlikely to happen.*

Example. *We apply the sampling algorithm to the same example of Figure 7.1a. Recall that the algorithm does not consider the cross-tree constraint. The algorithm starts at* $D_{\texttt{Sailboat}}$ *(the feature diagram rooted in the feature* `Sailboat`*). This node is a mandatory/optional node:*

- *A sub-configuration is sampled in the mandatory child (*$D_{\texttt{Hull}}$*). This child is an alternative group, so only one child is selected. All the children have the same variation degree (equal to 1), they are all likely to be selected.*
  - *Suppose that child* $D_{\texttt{Mono}}$ *is chosen. This child is a leaf node, so the sub-configuration returned is* $\{\texttt{Mono}\}$*.*

  The feature `Hull` *is added, so the sub-configuration returned is* $\{\texttt{Hull}, \texttt{Mono}\}$*.*
- *With probability* $\frac{3}{4} = 1 - \frac{1}{|Sols(D_{\texttt{Spi}})|}$ *the sub-feature diagram* $D_{\texttt{Spi}}$ *is sampled. Let's suppose that this probability is met. The* $D_{\texttt{Spi}}$ *feature diagram is an* `or` *node:*
  - *With probability* $\frac{1}{2}$ *the child* $D_{\texttt{Radial}}$ *is sampled. We suppose that this event does not happen, and that* $D_{\texttt{Radial}}$ *is not chosen, hence the sub-configuration returned is* $\{\}$*.*
  - *With probability* $\frac{1}{2}$ *the child* $D_{\texttt{Asym}}$ *is sampled. We suppose that this event does not happen, and that* $D_{\texttt{Asym}}$ *is chosen, so the sub-configuration returned is* $\{\texttt{Asym}\}$*.*

  We construct the sub-configuration $C = \{\} \cup \{\texttt{Asym}\} \neq \emptyset$*, hence the returned sub-configuration is* $\{\texttt{Spi}, \texttt{Asym}\}$*.*

The final configuration returned is $\{\texttt{Hull}, \texttt{Mono}, \texttt{Spi}, \texttt{Asym}, \texttt{Sailboat}\}$ *(union of the sub-configurations of the mandatory and the optional children plus the root node).*

7.7 Experimental Results

This section describes our experiments. First, the methodology is presented in Section 7.7.1 (implementation details, benchmark and state-of-the-art approaches tested). Section 7.7.2 contains the comparison with the other approaches in terms of t-wise coverage and running time. Finally, Section 7.7.3 confirms that the approach behaves the same way for higher values of t.

7.7.1 Methodology

Implementation

Our implementation is available online[1]. It is written in `Java`, using the CP solver `choco-solver` version 4.10.10 [46]. For the commonalities φ we used the linear time approximation presented in the previous Section 7.6.2. FrequencyDiff and RandomSearch are implemented using `choco-solver`'s search strategies. After each solution the search is restarted and the solution is excluded. A restart strategy is used when too many fails are encountered. The number of fails to restart follows a Luby sequence [178] of factor 50.

We compared these strategies with three state-of-the-art approaches:

- Baital[2] with 5 and 10 rounds. We use strategy 4 presented in [123] (numbered strategy 5 in the implementation) as it is among the best strategies in terms of coverage, and also among the fastest as it does not need to compute the set of combinations.
- Uniform sampling. We use Baital with 1 round for convenience, which is equivalent to using the uniform sampler WAPS [80].
- TableSampling [2] using the implementation in `choco-solver`. TableSampling takes 3 parameters as input: κ the pivot values for the number of solutions enumerated at each step, v the number of variables in the table, and p the probability of keeping a tuple in the table. In Chapter 5, we recommended using $\kappa = 1/p$, and values of v depending of the allowed running time and desired randomness. We use the sets of parameters $(\kappa, v, p) \in \{(4, 4, 1/4), (8, 6, 1/8), (16, 8, 1/16)\}$.

The experiments were run on single threads on a Xeon E7-8870 v4 / 20c / 1.4GH processor. For each instance, 100 solutions are generated. This solution generation is run

1. `https://github.com/MathieuVavrille/frequency-diff`
2. `https://github.com/meelgroup/baital`

twice with different random seeds, and the results in terms of running time or size of the coverage are averaged (using the arithmetic mean) over these two runs.

Benchmark

The instances used to test our approach come from the `uvl-models` [3] repository, in the UVL input format [142]. It contains feature models from various domains (e.g. automotive, operating systems,etc).

BAITAL takes as input a CNF formula, we used FeatureIDE [4] to transform the UVL format into `.dimacs` files. Instances where the conversion did not terminate (mostly due to the size of the instance) or raised an error (mostly due to bad naming of features that could not be easily fixed) were excluded from the benchmark. On two instances, BAITAL did not generate a configuration containing all the features due to an issue in the compilation of the d-DNNF representation. These two instances were also removed from the benchmark.

In the end, we applied the approaches to 123 instances. A large part of the instances (116 instances) come from the same initial benchmark [136]. These instances have between 1178 and 1408 features, and between 816 and 956 cross-tree constraints.

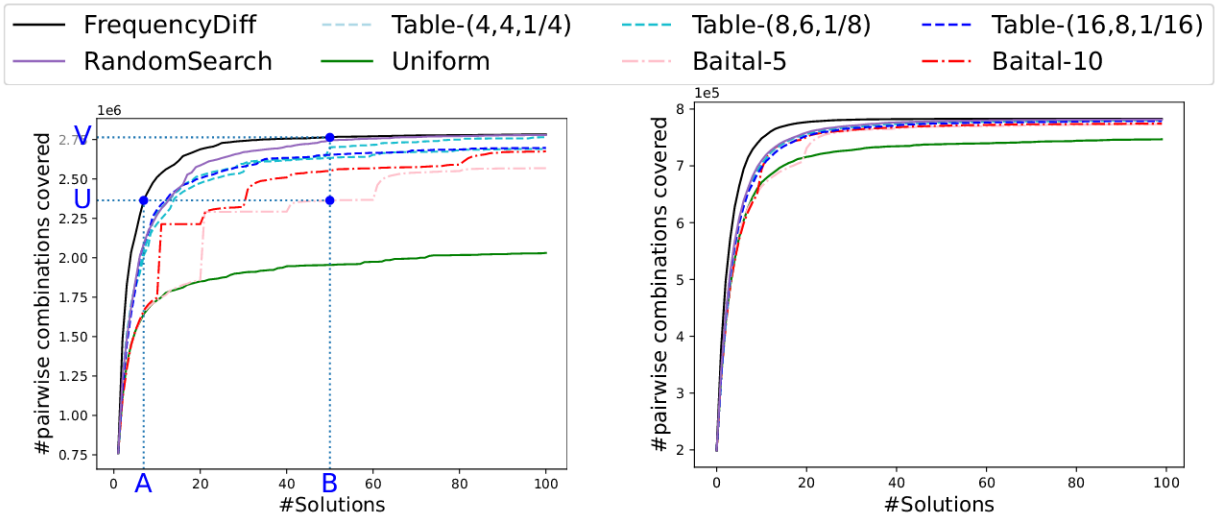
Evaluation Metrics

To evaluate the improvement of our strategy, we consider three metrics: the running time, the coverage, and the number of solutions to achieve a given coverage.

Speedup The running time is one of the most important metrics in some applications. We compute the speedup of our strategy over other approaches, i.e. how much faster our strategy could generate the required 100 solutions. Let τ^I_{freq} be the time taken by our approach to sample 100 solutions on the instance I, and τ^I_{baital} be the time taken by BAITAL. The speedup of our approach over BAITAL is then $\frac{\tau^I_{freq}}{\tau^I_{baital}}$ on instance I. We use the geometric mean to average the speedups over all instances in $\mathcal{I}$:

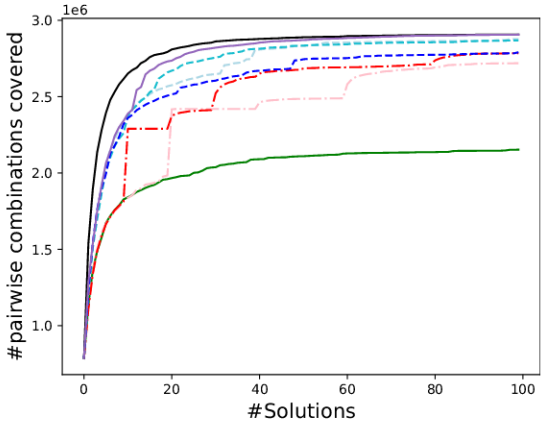
$$speedup = \left(\prod_{I \in \mathcal{I}} \frac{\tau^I_{freq}}{\tau^I_{baital}} \right)^{1/|\mathcal{I}|}$$

3. `https://github.com/Universal-Variability-Language/uvl-models`
4. `https://github.com/FeatureIDE/FeatureIDE`

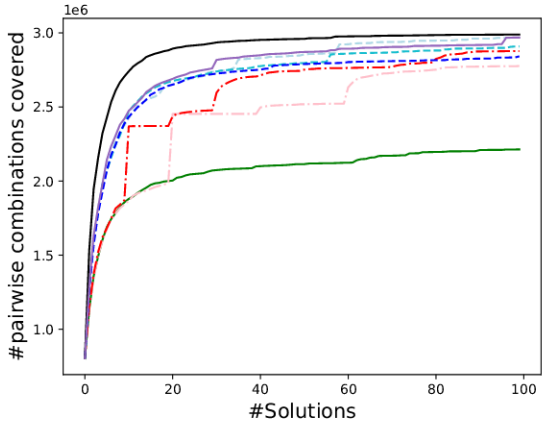


(a) Pairwise coverage on `brutus`

(b) Pairwise coverage on `busybox`



(c) Pairwise coverage on `grg`



(d) Pairwise coverage on `pc_vmWare`

Figure 7.3 – Evolution of the pairwise coverage on different instances

Coverage To see graphically what the metrics represent, we show in Figure 7.3a the evolution of the pairwise coverage, with different labels referring to points in the plots.

A first way to aggregate the results of all instances into a single value is to look at the coverage after a fixed number of solutions. Figure 7.3a shows the values behind the following description. In this figure, after 50 solutions (label B), Baital (with 5 rounds) covered 2,363,836 combinations (label U), and `frequency_diff` covered 2,763,767 combinations (label V). The ratio of improvement is $V/U = 1.17$, which means that after 50 solutions, `frequency_diff` covered 17% more combinations. We call this factor the *coverage improvement*. To aggregate the result, we use the geometric mean of these factors.

A second way to aggregate the coverage is to look at how many solutions the different

approaches need to generate to get the same coverage. After 50 solutions (label B), Baital covers 2,363,836 combinations (label U). FrequencyDiff covers the same number of combinations after only 7 solutions (label A). This means that FrequencyDiff can give the same coverage with $B/A \approx 7$ times fewer solutions than Baital. We call this ratio the *size improvement*. Again, we aggregate the results using the geometric mean.

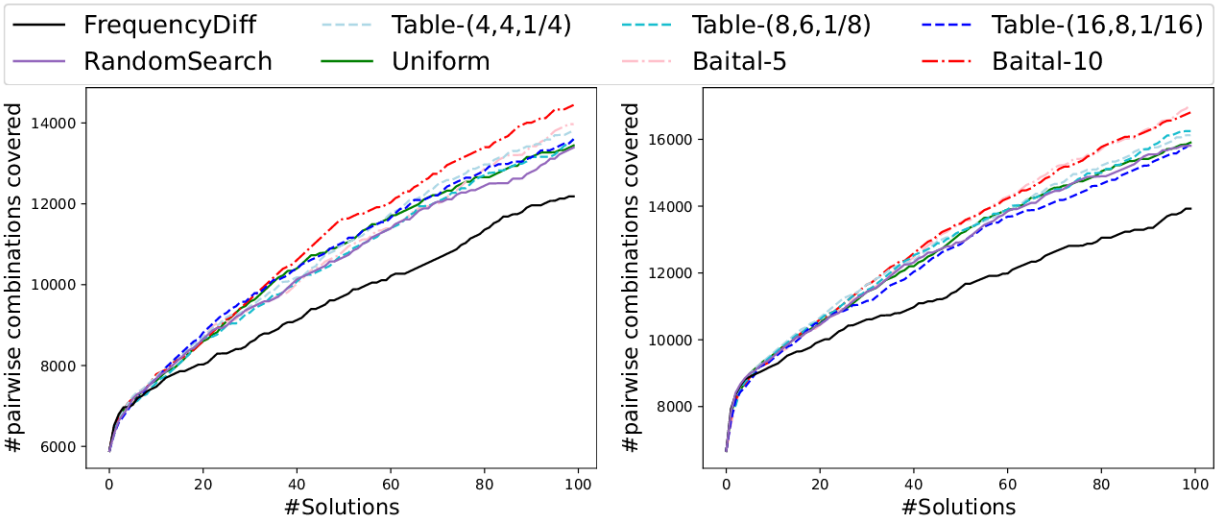
7.7.2 Comparison with other approaches

In this section, we compare the experimental results in terms of coverage and running time. All the the aggregated ratios (of coverage or running time) are summarised at the end of this section in Table 7.1.

Coverage

Figure 7.3 shows plots of the evolution of the number of pairwise combinations found by the different approaches. The behaviour is roughly the same for all instances, except for two specific instances that are discussed in the next sub-section. Apart from these two instances, our approach gives the best coverage on all but five instances (and RandomSearch often gives the second best coverage). On these five instances FrequencyDiff is the second best approach, just behind RandomSearch. This means that the search-strategies (RandomSearch and FrequencyDiff) outperform the other approaches on 121 instances (all the benchmark except the two particular instances). FrequencyDiff by itself outperforms *all* the other approaches on 116 out of the 123 instances of the benchmark. We can also look at the other approaches in detail.

- Uniform sampling is the worst approach in terms of coverage. It fails to find new combinations after the first few solutions and after 100 solutions, the coverage is much lower than all the other approaches. On average, FrequencyDiff finds 34% more combinations.
- Jumps in the coverage can be seen in the plots for Baital. These are due to the updating of the weights between the rounds. Before the first update, the curve follows the uniform sampling curve because all the weights are equal. At the end of the rounds, the weights are recomputed according to the solutions found, so the sampling is weighted towards features (and combinations) that have not yet been found, giving the jump in the coverage. On average, FrequencyDiff finds 4% more combinations than Baital-10.



(a) Instance `dm_ASEJ1` (b) Instance `dm_DissModel`

Figure 7.4 – Evolution of the pairwise coverage on two particular instances where our approach did not perform well.

- TableSampling performs better overall than Baital. It seems that the lower the number of variables in the tables, the better the coverage. This number of variables can bring the sampling closer or further away from uniformity. When v is high, the sampling is more uniform, so the coverage is closer to that of uniform sampling, which is the worse of all the approaches. When v is lower, TableSampling is closer to RandomSearch, so the coverage is better.

Analysis of Two Particular Instances

Of all the test suites generated, there were only two instances where FrequencyDiff did not outperform the other approaches. This section describes these two instances in detail.

Figure 7.4 shows the evolution of the pairwise coverage on these two instances. These plots are very different from the previous pairwise coverage plots in Figures 7.3 and 7.7. Instead of having a logarithmic-like growth (it becomes harder to find new combinations after some solutions), the two instances have a linear growth of the coverage. This means that each solution does not cover many new combinations.

To understand this behaviour, we show the feature model associated with the instance `dm_ASEJ1` in Figure 7.5. The other instance (`dm_DissModel`) is similar. This instance consists of 4 main features in an `or` group. The feature `Modes` has an `or` group with 4

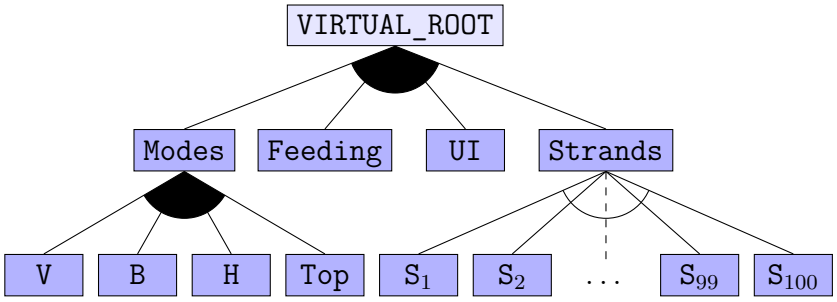


Figure 7.5 – Feature Model of the instance `dm_ASEJ1`. The feature's names have been simplified for clarity.

children. The special feature in this feature model is `Strands`. It has 100 children in an `alternative` group (only one of the children can be taken). It also contains a cross-tree constraint (the instance `dm_DissModel` does not have such a constraint).

The optimal strategy for optimising the t-wise coverage on this instance is to choose a different $\mathtt{S}_i$ on each solution. The reason why FrequencyDiff does not perform well is because of the very low commonality (i.e. frequency) of features $\mathtt{S}_i$ (around 1/100) combined with the alternative group they are in. This increases the chances of selecting the same feature more than once.

Example. *We show how the search may behave. We suppose that we have already sampled 20 solutions with FrequencyDiff, all containing a different $\mathtt{S}_i$. We want to estimate the probability of picking an already chosen $\mathtt{S}_i$. The observed frequency of the already chosen $\mathtt{S}_i$ is* $1/20$*. For the already chosen $\mathtt{S}_i$, the weights are then* $1/20 - 1/100 \approx 1/20$*, and is only* $1/100$ *for the not yet chosen $\mathtt{S}_i$. Then the probability of choosing an already taken $\mathtt{S}_i$ is more than* $1/2$ *(it is* $\frac{20 \cdot \frac{1}{20}}{20 \cdot \frac{1}{20} + 80 \cdot \frac{1}{100}} = \frac{1}{1.8}$*). By default, the value chosen would be 0, as the observed frequency is higher than the commonality, but there is a* $\frac{1+\frac{1}{20}}{2}$ *chance of swapping this value, thus adding this feature to the solution.*

We argue that the modelling of this instance as a feature model was not the right way to represent it. Therefore, the use of feature models based approaches is not appropriate. The `Strands` feature with 100 children S_i should have been modelled as an integer variable (for example in a CP framework) as $\mathtt{Strands} \in \{1, \ldots, 100\}$. Such modelling of integer variables (called attributes) in feature models has been studied in [135] and allows the use of CP's global constraints. RandomSearch would work as is in a CP framework (where

the values are not necessarily binary). If necessary, it would also be possible to prevent completely already chosen $\mathtt{S}_i$ from being taken. The fact that FrequencyDiff does not perform well on these two instances is not representative of the performance on the whole benchmark set.

Size of the Solution Set

The improvements in coverage may seem small, but as the number of combinations found increases, it becomes harder and harder to find new combinations. Therefore, even a small percentage of improvement is hard to achieve when the coverage is already high. Due to the inverse exponential behaviour of the coverage, the number of solutions needed to achieve the same coverage varies greatly for different approaches.

On average, FrequencyDiff requires 5 time fewer solutions than Baital-10 to achieve the same coverage. This means that, on average, the coverage of FrequencyDiff will be the same after 20 solutions as that of Baital-10 after 100 solutions.

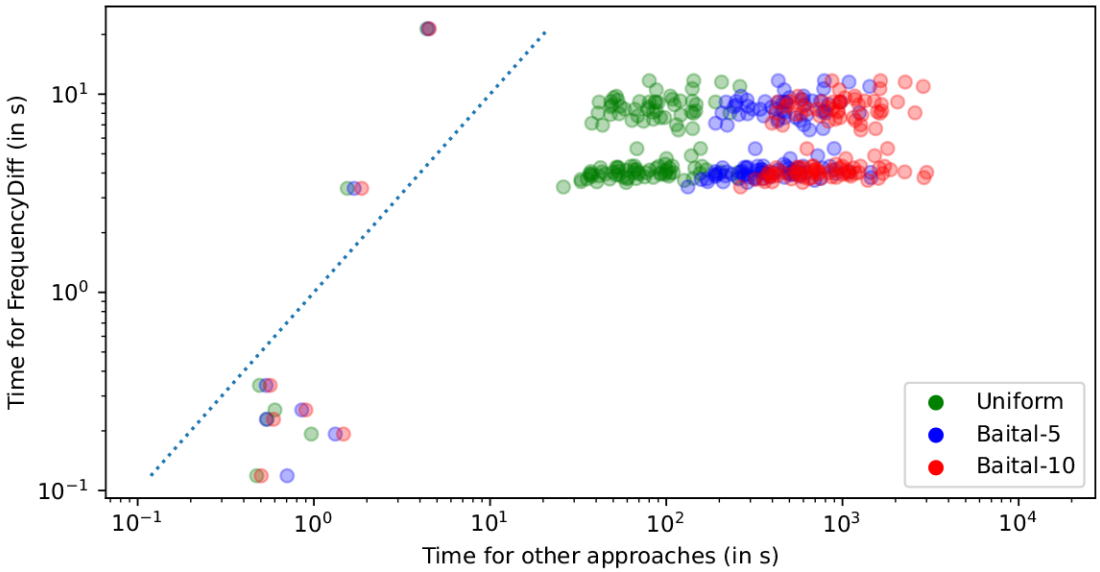
The coverage of TableSampling is better than that of Baital, and this reflects to the number of solutions needed to obtain a given coverage. To obtain the same coverage as TableSampling after 100 solutions, FrequencyDiff requires 2.4 times fewer solutions.

To achieve the same coverage as RandomSearch (the second best approach), FrequencyDiff requires 1.4 times fewer solutions.

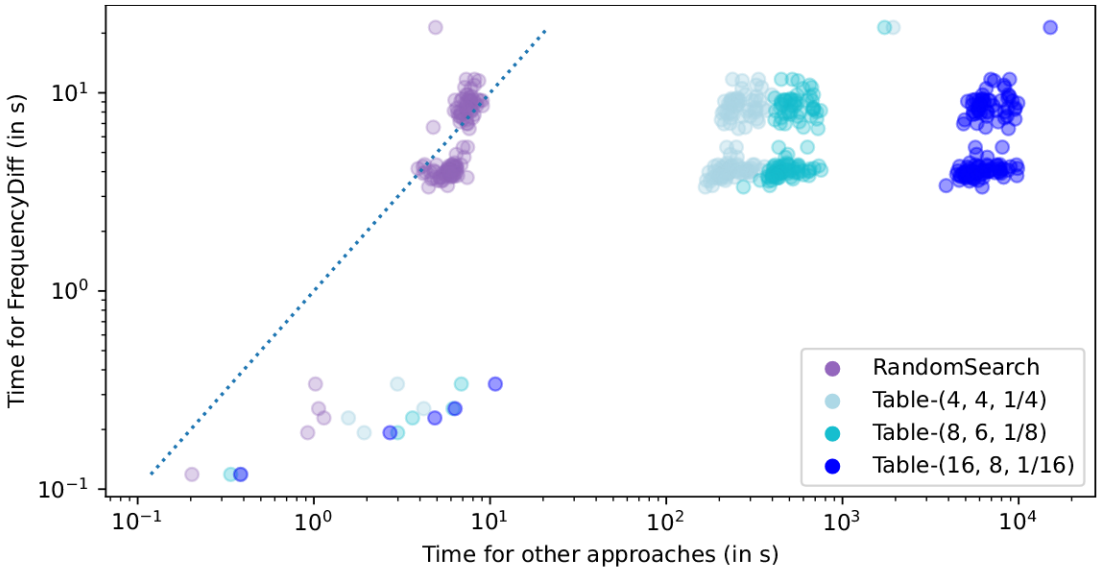
Remember that the goal of generating solutions is to use them as tests of a product line. The actual implementation of the solution in the product line could be expensive (in the case of an automotive product line) or time consuming (in the case of software compilation). Any reduction in the number of solutions generated, without affecting the coverage, is directly reflected in faster or lower cost of tests of the product line. Our strategy FrequencyDiff significantly improved this size of test suite generated.

Running Time

A scatter plot of the running time of the other approaches compared to `frequency_diff` is given in Figure 7.6. For clarity, the plot is divided into a comparison with SAT sampling approaches (uniform sampling and Baital) in Figure 7.6a and CP sampling approaches (RandomSearch and TableSampling) in Figure 7.6b. On the two scatter plots, the y-axis is the time taken by FrequencyDiff. If a point (an instance) is below the dotted line, it means that FrequencyDiff was faster.



(a) Compared to BAITAL and uniform sampling



(b) Compared to TABLESAMPLING and RANDOMSEARCH

Figure 7.6 – Scatter plot of running times. Each dot is an instance.

FREQUENCYDIFF is the fastest of all approaches overall. RANDOMSEARCH can be considered to be as fast as FREQUENCYDIFF. All the other approaches are orders of magnitude slower than the search strategy approaches. In only two instances was FREQUENCYDIFF slower than uniform sampling and BAITAL.

FREQUENCYDIFF has a speedup of 12.27 over uniform sampling. Most of the time of BAITAL is spent computing the weights of the d-DNNF representation, which is done

at the beginning of each round (and only once in the case of uniform sampling). This is reflected in the running time, as Baital-5 is about 5 times slower than uniform sampling, and Baital-10 is about 10 times slower.

FrequencyDiff has a speedup of 41 compared to TableSampling with parameters $(4, 4, \frac{1}{4})$. With parameters $(16, 8, \frac{1}{16})$, TableSampling is the slowest approach. The running time increases significantly when the number of variables increases, because more tuples have to be generated during table creation. As noted in the previous section, the coverage is better when v is smaller. We can conclude that there is no reason (on the problem of t-wise coverage) to use high values of v.

Analysis of TableSampling

The analysis of TableSampling is interesting and underlines what we want to show in this chapter. In Chapter 5 we showed that the sampling becomes more uniform as v increases. However, in the t-wise coverage experiments, we show that it is better to have a small value of v to get a good coverage. In fact, we have shown that for the task of t-wise coverage, uniformity is a disadvantage and not an advantage. In the case of TableSampling, using a high value for v makes the sampling more uniform, and therefore is a disadvantage.

The experiments also show opposite results to Chapter 5 in terms of running time compared to RandomSearch. In Chapter 5 the benchmark was made from hard instances of the MiniZinc challenge. The search strategy was very important to be able to find solutions quickly, and RandomSearch made bad decisions. On feature models it is much easier to find a solution, so there is no downside to using RandomSearch as a search strategy (the propagation removes most of the inconsistent values).

Summary of the experiments

We summarise the coverage and running time results given in the previous sections. Table 7.1 gives all the ratios (compared to FrequencyDiff) that were partly mentioned in the previous sections. When a coverage ratio was mentioned in the previous section, it referred to the coverage after 100 solutions. It is also possible to give this ratio after only 50 solutions, and these ratios are present in the table with the label 50. These ratios do not differ much between those computed after 100 solutions. The running time is only recorded at the end of the 100 solution generation.

Table 7.1 – Summary of the coverage/running time average ratios between other approaches and FrequencyDiff.

	RandomSearch	Uniform Sampling	Baital		TableSampling $-(\kappa, v, p)$		
			5	10	$(4, 4, \frac{1}{4})$	$(8, 6, \frac{1}{8})$	$(16, 8, \frac{1}{16})$
Coverage-100	1.00	1.34	1.07	1.04	1.01	1.02	1.04
Coverage-50	1.01	1.37	1.16	1.07	1.03	1.04	1.06
Size-100	1.39	23.89	7.44	5.10	2.37	3.65	5.21
Size-50	1.50	14.21	6.98	4.12	2.25	2.88	3.58
Time Speedup	1.18	12.27	60.35	118.29	41.14	82.83	961.99

It should be noted that all the ratios compared to FrequencyDiff in this table are greater than 1. This means that overall on the whole benchmark, in coverage or running time, FrequencyDiff is the best strategy compared to the other approaches tested.

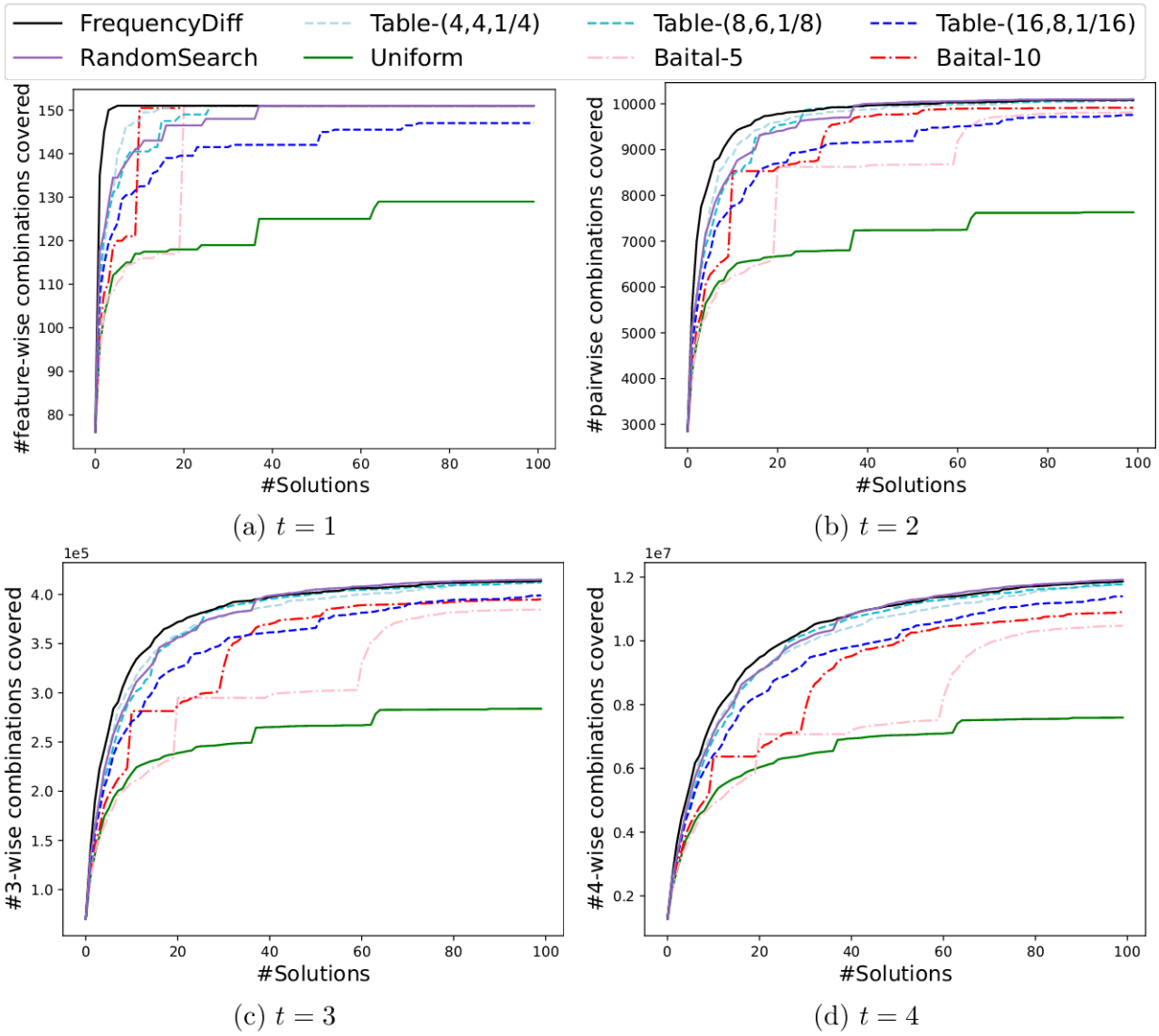
7.7.3 Higher Value of t

In the previous section we plotted the evolution of the pairwise coverage. Ideally we would like to evaluate the t-wise coverage for higher values of t, but this is too expensive to compute. Most of the instances have thousands of features, so there are more than a billion 3-wise combinations covered by each solution. However, we can compute the exact number of 3-wise (and even 4-wise) combinations on a smaller instance, or approximate this value using ApproxCov [124] for larger instances.

Exact Computation

We plot in Figure 7.7 the evolution of the t-wise coverage for $t \in \{1, 2, 3, 4\}$ on the instance `berkeleydb`. We can see that the behaviour of the approaches remains the same for all values of t.

We can see that Baital performs better for $t = 1$. This is a consequence of the literal-weight function used as the distribution weight. This function assigns weights to the features, and samples according to these weights. Hence, we see that after updating the weights once (at solution 10 for Baital-10, and solution 20 for Baital-5) all the feature-wise combinations are found.



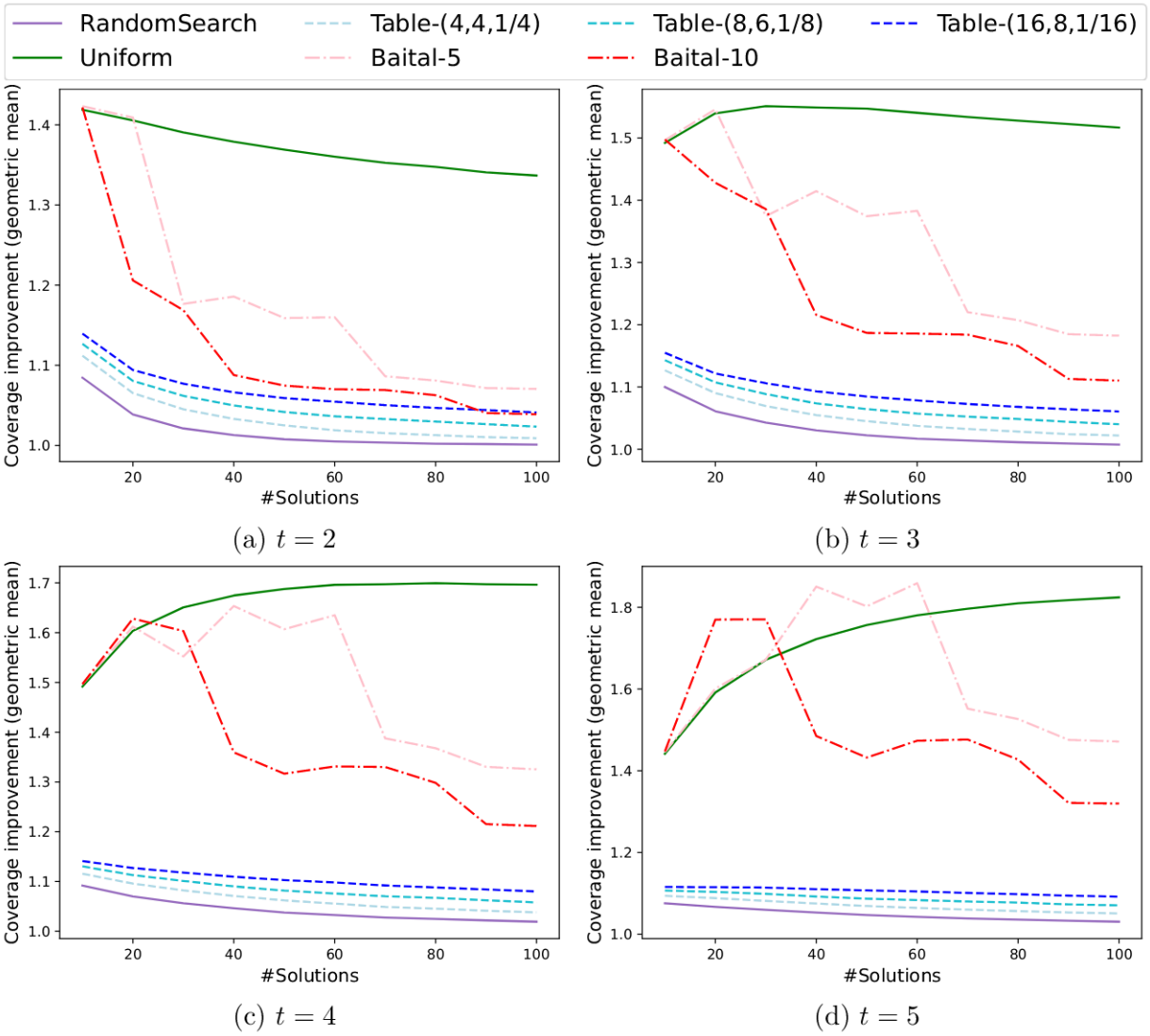
(a) $t = 1$ (b) $t = 2$

(c) $t = 3$ (d) $t = 4$

Figure 7.7 – Evolution of the t-wise coverage on the `berkeleydb` instance, with different values of t.

Approximate Computation

In the instance `berkeleydb`, the exact number of combinations can be computed for higher values of t because there are few features. However, this is impossible for larger instances. ApproxCov [124] is an algorithm for computing an approximation of the number of combinations covered by a test suite. Here we use this tool to get an approximation of the behaviour of our approach compared to the other approaches. We use ApproxCov with default parameters, i.e. with a probability of 0.95, the result is within a factor of 1.05 of the exact value.



(a) $t = 2$ (b) $t = 3$

(c) $t = 4$ (d) $t = 5$

Figure 7.8 – Evolution of the improvement of FrequencyDiff in t-wise (approximate) coverage size, with different values of t. For a given number of solutions, we plot the (geometric) average of the improvement in coverage size compared to FrequencyDiff. For example, for $t = 3$, after 60 solutions, FrequencyDiff covers on average 1.2 times more combinations than Baital-10.

In Figure 7.8 we plot the average improvement of the coverage size over all instances, for different numbers of solutions (by restricting the generated test suite to the first n configurations, for $n \in \{10, 20, \ldots, 90, 100\}$). We can see that for all values of $t \in \{2, 3, 4, 5\}$, FrequencyDiff generates test suites that cover more combinations than the other approaches (all the coverage improvements are greater than 1). This confirms the fact that our approach can generate good coverage test suites for higher values of t,

even for large instances.

In the previous section, we only showed the pairwise coverage. As noted in [133], a test suite with a good t-wise coverage will most likely also have good $t+1$-wise coverage. We have proved this experimentally. Thus, the pairwise coverage results we showed in the previous section extend to higher values of t.

7.8 Conclusion

In this chapter we showed that CP's search strategies are an excellent way of generating test suites with high t-wise coverage. We explained this result by analysing RandomSearch's probability of returning a solution containing a given combination. This analysis showed that RandomSearch is more suited to generate test suites with high t-wise covering than uniform sampling.

We proposed an improvement to RandomSearch called FrequencyDiff, which uses information about previously generated solutions. Using this information, and comparing it with the commonality of the features, it tweaks the distribution used in RandomSearch to favour interesting features.

We experimentally tested these search strategies on feature models with more than a thousand features. The results showed that the search strategies outperformed other sampling approaches by several orders of magnitude in running time, significantly improved the coverage, and reduced the number of solutions required to achieve a given coverage. FrequencyDiff also improves the coverage over RandomSearch, making it the best of the approaches we tested, with no running time overhead.

Conclusion

In this part, we used search strategies to extract a good subset of solutions to constrained problems. We showed that the default random search strategy, RandomSearch, produces good sets of solutions. We proposed to improve it by weighting the choice of the variable and value to orient the search towards interesting sub-spaces. This can greatly improve the quality (diversity, or number of combinations covered) of the solutions generated. We have also shown that uniform sampling is not always beneficial, because of how the solutions are distributed in the search space.

In this part, we evaluated the quality of a solution set in two different ways: first in pattern mining using the Jaccard index as a diversity measure, and second, by considering the number of combinations covered by the solutions. Depending on the application, there are several other ways to evaluate a solution set.

In the following part, we study diversity in more detail. We analyse how the choice of the distance and the aggregator impacts the algorithms. For different distances, we show propagation algorithms, approximation algorithms and prove diversity guarantees. We also study diversity in a multi-objective framework, showing state-of-the-art multi-objective optimisation algorithms and new approaches to finding good and diverse solutions.

Part IV

Questions of Diversity

Chapter 8

Back to the Definitions

This chapter contains questions of diversity, linking the mathematical definitions with the implementations as constraints and the properties of the algorithms. Part of this chapter was presented in the semi-formal conference organised by the doctoral school for PhD students (JDOC).

8.1 Introduction

Diversity bridges the gap between solvers, with the rigid backtrack-search, and users, who want to be presented with multiple diverse solutions. In this chapter, we consider diversity from three perspectives: how users define their diversity problems, how diversity constraints can be implemented in solvers, and what are the properties of the approximation algorithms.

Initially in CP, diversity was defined in [24]. The authors define the `MaxDiverseKSet` and `MostDistant` problems, and diversity constraints to solve these problems. These constraints are a bound on the minimum distance between all the solutions. However, to aggregate the distances, the sum is often used [44, 54]. In this chapter, we come back to the definitions of diversity in constrained problems, and we analyse in detail the definitions and properties of the diversity constraints.

First, in Section 8.2 we revisit the definitions to properly define the diversity problems. This defines the diversity from the user's point of view. In particular, we allow the user to specify an aggregator for the pairwise distances. In Section 8.3, we look at the diversity constraints from a solver's perspective. We are interested in the complexity of the propagation of the constraint, and show reformulations of the diversity constraints either as smaller constraints or as existing global constraints. Experimentally, the behaviour of the aggregators (and their differences) is presented in Section 8.4. In Section 8.5, we analyse the guarantees of approximation algorithms. We prove that, in the worst case, the minimum distance between the solutions returned by approximation algorithms is *at*

most twice worse than that of the optimal solution set. We also analyse the behaviour of uniform sampling in a simple case.

8.2 Diversity: Revisiting the Definitions

In this section, we revisit the definitions of the diversity problems presented in Chapter 2 Section 2.6. These definitions consider multiple pairwise distances. To get a single value to compare the diversity quality of solution sets, all the distances are aggregated into a single value.

In the original definition [24], the distances are aggregated using the minimum. Other articles [44, 54] use the definition of these problems but use the sum to aggregate the distances. In [28], the authors leave the choice to the user modelling the problem.

Remark. *The choice of the* $\sum$ *or* min *aggregator has an interesting parallel with computational social choice [165] and multi-agent resource allocation [166] under the notions of* utilitarian *and* egalitarian *social welfare. In a utilitarian setting, the* total satisfaction *of agents should be maximised, but some agents may not be satisfied. In an egalitarian setting, the satisfaction of* the least satisfied *agent should be maximised. Other measures (i.e. aggregators) are proposed in this area, such as the* Nash product *(multiplication of the distances), which is a compromise between the utilitarian and the egalitarian social welfare, or the* median rank dictators*, where the median of the distances is maximised.*

We now redefine the `MaxDiverseKSet` and `MostDistant` problems but with an arbitrary aggregator $\mathcal{A}$.

Definition 11' (`MaxDiverseKSet`)**.** Let $k \geq 2$ be an integer and $\mathcal{P}$ be a CSP, with solutions $Sols(\mathcal{P})$, δ be a distance over these solutions, and $\mathcal{A}$ an aggregator of distances (min or $\sum$). `MaxDiverseKSet`(k) is the problem of finding a subset of solutions $\widetilde{S} \subseteq Sols(\mathcal{P})$ of size k that maximises the distances between the solutions, i.e.

$$\widetilde{S} = \underset{\substack{S \subseteq Sols(\mathcal{P}) \\ |S|=k}}{\operatorname{argmax}} \underset{\substack{s,s' \in S \\ s \neq s'}}{\mathcal{A}} \delta(s, s') \,.$$

Definition 12' (`MostDistant`)**.** Let $S \in Sols(\mathcal{P})$ be a set of solutions, δ be a distance and $\mathcal{A}$ be an aggregator. `MostDistant`(S) is the problem of finding the solution $\widetilde{s}$ that is

most distant from all the solutions in S, i.e.

$$\widetilde{s} = \underset{s \in Sols(\mathcal{P})}{\operatorname{argmax}} \underset{s' \in S}{\mathcal{A}} \delta(s, s') \,.$$

Remark. *When the number of solutions is fixed (to k), aggregating all the distances using the sum is equivalent to computing the average distance (by a factor of $k(k-1)/2$ for* `MaxDiverseKSet` *or k for* `MostDistant`*).*

In Chapter 2.6 Section 2.6.2 we already presented how a constraint satisfaction problem can be reformulated as a constraint optimisation problem to solve `MaxDiverseKSet` or `MostDistant`. For `MaxDiverseKSet` this is done by duplicating the initial problem k times using sets of variables $\mathcal{X}^1, \ldots, \mathcal{X}^k$ and adding a constraint on the distance between the solutions of the duplicated problems (i.e. between the sets $\mathcal{X}^1, \ldots, \mathcal{X}^k$). For `MostDistant`, the distance constraint between the problem and the previous solutions can be added. We formally define the diversity constraints, i.e. the constraints on the distances between the variables (and the solutions in the case of `MostDistant`).

Definition 38 (Diversity constraints)**.** Let δ be a distance function, $\mathcal{A}$ be an aggregator, and k be an integer. Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ and d be variables, and let $\mathcal{S}$ be a set of k solutions. The $\texttt{single\_diversity}_{\mathcal{A},\delta}$ constraint considers the distance from one set of variables to multiple solutions already found:

$$\texttt{single\_diversity}_{\mathcal{A},\delta}(\mathcal{X}, \mathcal{S}, d) \Leftrightarrow \underset{S \in \mathcal{S}}{\mathcal{A}} \delta(\mathcal{X}, S) \geq d \tag{8.1}$$

For $1 \leq j \leq k$, let $\mathcal{X}^j$ be k sets of n variables, and d be a variable. The $\texttt{multiple\_diversity}_{\mathcal{A},\delta}$ constraint considers the distance between the sets of variables $\mathcal{X}^1, \ldots, \mathcal{X}^k$, i.e. $\delta(\mathcal{X}^i, \mathcal{X}^j)$ is the distance between the i-th and the j-th duplicated solution:

$$\texttt{multiple\_diversity}_{\mathcal{A},\delta}\left(\mathcal{X}^1, \ldots, \mathcal{X}^k, d\right) \Leftrightarrow \underset{1 \leq i < j \leq k}{\mathcal{A}} \delta\left(\mathcal{X}^i, \mathcal{X}^j\right) \geq d \tag{8.2}$$

Remark. *In [24], the* $\texttt{single\_diversity}_{\min,\delta_{\mathcal{H}}}$ *constraint is called* $\texttt{Diverse}_{\min}$ *and the* $\texttt{single\_diversity}_{\Sigma,\delta_{\mathcal{H}}}$ *constraint is called* $\texttt{Diverse}_{\Sigma}$*. A propagation algorithm for* $\texttt{single\_diversity}_{\Sigma,\delta_{\mathcal{H}}}$ *is also presented.*

In this definition, $\texttt{multiple\_diversity}_{\mathcal{A},\delta}$ is the diversity constraint for `MaxDiverseKSet` and $\texttt{single\_diversity}_{\mathcal{A},\delta}$ is the one for the `MostDistant` problem.

8.3 Analysis of Diversity Constraints

In the previous section we revisited the definitions of the diversity problems, and their associated constraints. Here we look at the diversity constraints in more detail, depending on the aggregator used.

8.3.1 $\sum$ Aggregator

Here we focus on the $\sum$ aggregator. When using the $\sum$ aggregator, all the pairwise distances are summed. The distances are computed between two vectors in n dimensions. We first remark that some distances are also computed by summing sub-distances on each dimension of the problem. We call such distances *separable* distances.

Definition 39 (Separable Distance)**.** A distance $\delta : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is *separable* iff there exists a function $\widetilde{\delta} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that for all $x, y \in \mathbb{R}^n$,

$$\delta(x, y) = \sum_{i=1}^{n} \widetilde{\delta}(x_i, y_i)\,.$$

Lemma 3. *The Hamming and Manhattan distances are separable, with*

$$\widetilde{\delta_{\mathcal{H}}}\,(a, b) = \mathbb{1}_{a \neq b} \tag{8.3}$$

$$\widetilde{\delta_{l_1}}\,(a, b) = |a - b|\,. \tag{8.4}$$

Using a separable distance means that each dimension is independent of the other. This allows us to reformulate the diversity constraints. We introduce the diversity constraints on a single dimension.

Definition 40. Let δ be a separable distance function and k be an integer. Let X and d be a variable, $s_1, \ldots, s_k$ be integers, and $X_1, \ldots, X_k$ be variables. We define the diversity constraints on a single dimension:

$$\texttt{single\_diversity\_dim}_{\delta}(X, s_1, \ldots, s_k, d) \Leftrightarrow \sum_{i=1}^{k} \widetilde{\delta}(X, s_i) \geq d \tag{8.5}$$

$$\texttt{multiple\_diversity\_dim}_{\delta}(X_1, \ldots, X_k, d) \Leftrightarrow \sum_{1 \leq i < j \leq k} \widetilde{\delta}(X_i, X_j) \geq d \tag{8.6}$$

Remark.

- *The constraints only enforce the inequality with the variable d. It is possible to constrain the equality, but it is not necessary, because d will be maximised. So even if d is loosely constrained, it will be instantiated to the largest value possible in its domain to maximise the diversity.*
- *The* `single_diversity_dim`$_\delta$ *constraint is binary, i.e. the scope contains only two variables.*

These two constraints enforce diversity on a single dimension of the problem. When using a separable distance, the dimensions are independent of each other, so we can use these constraints to reformulate the main diversity constraints.

Proposition 5. *For δ a separable distance function,*

$$\texttt{single\_diversity}_{\Sigma,\delta}(\mathcal{X}, \mathcal{S}, d) \Leftrightarrow \begin{cases} \forall i \in \{1, \dots, n\}, \texttt{single\_diversity\_dim}_\delta(X_i, S_1[i], \dots S_k[i], d_i) \\ \sum_{i=1}^{n} d_i \geq d \end{cases} \tag{8.7}$$

$$\texttt{multiple\_diversity}_{\Sigma,\delta}(\mathcal{X}^1, \dots, \mathcal{X}^k, d) \Leftrightarrow \begin{cases} \forall i \in \{1, \dots, n\}, \texttt{multiple\_diversity\_dim}_\delta(X_i^1, \dots, X_i^k, d_i) \\ \sum_{i=1}^{n} d_i \geq d \end{cases} \tag{8.8}$$

Proof. We use the definitions of the constraint:

$$\begin{aligned} \texttt{single\_diversity}_{\Sigma,\delta}(\mathcal{X}, \mathcal{S}, d) &\Leftrightarrow \sum_{j=1}^{k} \delta(\mathcal{X}, S_j) \geq d && \text{(by definition)} \\ &\Leftrightarrow \sum_{j=1}^{k} \sum_{i=1}^{n} \widetilde{\delta}(X_i, S_j[i]) \geq d && (\delta \text{ separable}) \\ &\Leftrightarrow \sum_{i=1}^{n} \sum_{j=1}^{k} \widetilde{\delta}(X_i, S_j[i]) \geq d && \text{(sum permutation)} \end{aligned}$$

We create the variables d_i (with the same domain as d) and add the constraints

$$\sum_{j=1}^{k} \widetilde{\delta}(X_i, S_j[i]) \geq d_i$$

. The reformulation is then

$$\sum_{i=1}^{n}\sum_{j=1}^{k}\widetilde{\delta}(X_i, S_j[i]) \geq d \Leftrightarrow \begin{cases} \forall i \in \{1,\ldots,n\}, \sum_{j=1}^{k}\widetilde{\delta}(X_i, S_j[i]) \geq d_i \\ \sum_{i=1}^{n} d_i \geq d \end{cases}$$
$$\Leftrightarrow \begin{cases} \forall i \in \{1,\ldots,n\}, \texttt{single\_diversity\_dim}_{\delta}(X_i, S_1[i], \ldots S_k[i], d_i) \\ \sum_{i=1}^{n} d_i \geq d \end{cases}$$

□

Reformulating by using smaller constraints may not always be a good idea, as we know from the example of the `alldifferent` constraint (some values may be arc consistent with the network of disequalities, but not arc consistent with the `alldifferent` constraint). However, in our case we do not alter the quality of the propagation.

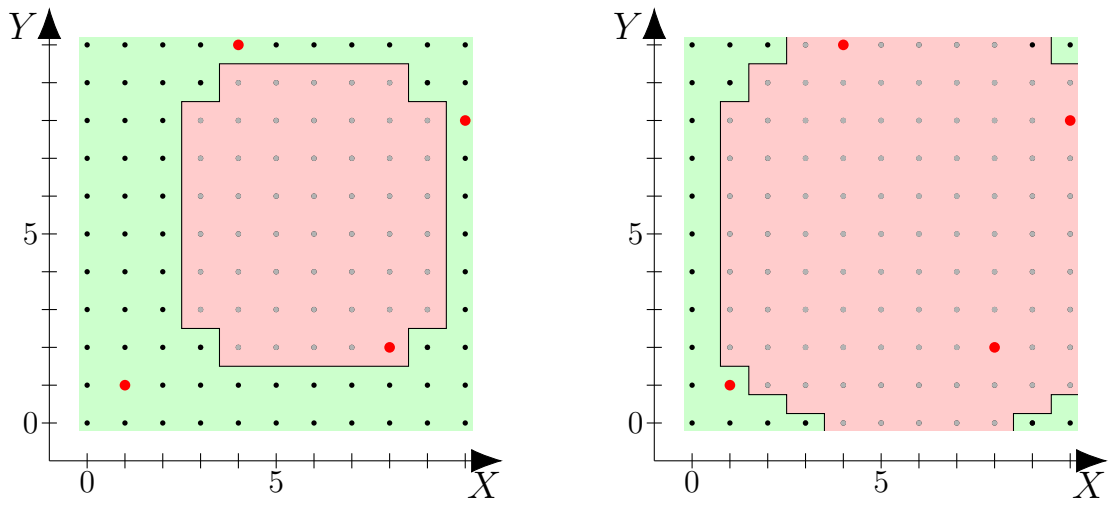
Proposition 6. *All the values are arc consistent on the constraint* $\texttt{single\_diversity}_{\Sigma,\delta}$ *(resp.* $\texttt{multiple\_diversity}_{\Sigma,\delta}$*) iff the network of constraints of the reformulation in equation 8.7 (resp. equation 8.8) is arc consistent.*

Proof. In Appendix A.3.1. □

This reformulation does not lose any propagation power because the structure of the constraint network is a tree (d at the root, linked to the variables d_i, which are linked to the variables X_i). This should be constrasted with the clique structure of `alldifferent` constraint reformulation (there is a constraint $x_i \neq x_j$ between any pair of variables x_i and x_j).

Example. *Figure 8.1 shows an example of the space allowed by a* $\texttt{single\_diversity}_{\Sigma,\delta_{l_1}}$ *on a problem with two variables X and Y with domain $\{0,\ldots,10\}$ and no constraints. The solutions $\mathcal{S}$ found so far are $(1,1),(4,10),(8,2)$ and $(10,8)$, marked by red dots in the figure. The green area shows all the possible solutions, and the red area shows the search space that contain no solutions. In Figure 8.1a a value of $d = 31$ is used. Using this value, nothing can be propagated by arc consistency, because we see that $X = 0$ is a support for all possible values $\mathcal{D}(Y)$, and $Y = 0$ is a support for all possible values in $\mathcal{D}(X)$.*

Using $d = 35$, the values $x \in \{4,\ldots,8\}$ can be removed from $\mathcal{D}(X)$ because there is no value $y \in \mathcal{D}(Y)$ such that (x,y) satisfies the constraint. We can look at the reformulation



(a) With $d = 31$ (b) With $d = 35$

Figure 8.1 – Example of propagation of the $\texttt{single\_diversity}_{\Sigma,\delta_{l_1}}((X,Y),\mathcal{S},d)$ constraint for $\mathcal{S} = \{(1,1),(4,10),(8,2),(10,8)\}$. The green area contains all the solutions.

of the constraint using Proposition 5. This reformulation is

$$\textit{single\_diversity}_{\Sigma,\delta_{l_1}}((X,Y),\{(1,1),(4,10),(8,2),(10,8)\},d)$$
$$\Leftrightarrow \begin{cases} |X-1|+|X-4|+|X-8|+|X-10| \geq d_1 \\ |Y-1|+|Y-2|+|Y-8|+|Y-10| \geq d_2 \\ d_1 + d_2 \geq d \end{cases}$$

The maximum value for d_2 is 21 (for $Y = 0$). For $d = 35$ this means that d_1 is necessarily greater than or equal to 14. For all the values of $X \in \{4,\ldots,8\}$, d_1 is lower than or equal to 13, so these values can be removed from the domain of X.

In this example, we see that the reformulation is able to remove as many values as the initial constraint, as guaranteed by Proposition 6.

We would also like to point out that the most distant point is $(0,0)$. This may seem counter-intuitive, because it is very close to $(1,1)$, but it is also very far from all the other points. Because the $\sum$ aggregator is used, all the distances are taken into account, without any minimum distance constraint. This fact will also be shown in Section 8.4.

We now go one step further and look at how the diversity constraints on each dimension can be propagated. We first look at the $\texttt{single\_diversity\_dim}_{\delta}$ constraint, then we consider two cases of $\texttt{multiple\_diversity\_dim}_{\delta}$, with $\delta = \delta_{\mathcal{H}}$ or $\delta = \delta_{l_1}$.

1 **Function** $\textsc{Propagate}_{\texttt{single\_diversity\_dim}}(X, s_1, \ldots, s_k, d, \delta)$
  Data: Two variables X and d, integers s_i, and a distance δ.
  Result: The domains of X and d reduced by arc consistency.
2 $M \leftarrow \max_{v \in \mathcal{D}(X)} \sum_{i=1}^{n} \widetilde{\delta}(v, s_i)$
3 $\mathcal{D}(d) \leftarrow \mathcal{D}(d) \cap [-\infty, M]$ // d upper bounded by M
4 **for** $v \in \mathcal{D}(X)$ **do**
5 **if** $\sum_{i=1}^{n} \widetilde{\delta}(v, s_i) < \min \mathcal{D}(d)$ **then**
6 Remove v from $\mathcal{D}(X)$

Algorithm 8.1: Propagation of $\texttt{single\_diversity\_dim}_{\delta}$.

Case of $\texttt{single\_diversity\_dim}_{\delta}$

The $\texttt{single\_diversity\_dim}_{\delta}$ constraint is a binary constraint, with the two variables X and d and parameters $s_1, \ldots, s_k$. Algorithm 8.1 presents a procedure for propagating arc consistency for the constraint. This procedure is adapted from the bound consistency propagator of a `sum` constraint [22]. First, the maximum of the sum of the distances between the values of X and the s_i are computed. This value M is a bound of the variable d, so we can reduce its domain. Then, for all variables v in the domain $\mathcal{D}(X)$ of X, if the sum of the distances between v and the s_i is less than the minimum value of d (line 5), then we can safely remove v from $\mathcal{D}(X)$.

The values $\sum_{i=1,\ldots,n} \widetilde{\delta}(v, s_i)$ for all $v \in \mathcal{D}(X)$ can be pre-computed in time $\mathcal{O}(|\mathcal{D}(X)|+k)$ when using the Hamming or Manhattan distance (if the s_i are sorted). For the Hamming distance, it is only necessary to count how many s_i take a value v (as in the counting sort). For the Manhattan distance, we can use Proposition 8 and compute the distances incrementally.

Another improvement is to loop over the values of X in increasing order of $\sum_{i=1}^{n} \widetilde{\delta}(v, s_i)$ in line 4. This way, if the condition in line 5 is not satisfied, we know that none of the following values can be removed. Overall, this propagator makes a linear number of computations from the root node of the search to the bottom of the search tree. However, this does not mean that it has a constant amortized complexity, since the depth of the search tree is unknown.

Remark. *The propagation algorithm presented requires very few computations to ensure arc consistency. Here, we refer to other implementations of the* $\texttt{single\_diversity\_dim}_{\delta}$ *constraint.*

- *When using the Hamming distance,* $\texttt{single\_diversity\_dim}_{\delta_{\mathcal{H}}}$ *is equivalent to a* `count` *constraint:*

$$\texttt{count}(X, \{s_1, \ldots, s_k\}, \leq, k-d) \Leftrightarrow |\{s_i \mid X = s_i\}| \leq k-d\,.$$

- *It is possible to use smart tables [35] to represent the constraint:*

$$\texttt{smart\_table}\left((X,d), \left\{\left(v, \cdot \leq \sum_{i=1}^{n} \widetilde{\delta}(v, s_i)\right) \mid v \in \mathcal{D}(X)\right\}\right)\,.$$

  This smart table has a size linear in the domain of X*. It could also be implemented as a classical table, by enumerating the possible values of the smart table (of size at worst* $\mathcal{O}(|\mathcal{D}(X)| \cdot |\mathcal{D}(d)|)$*):*

$$\texttt{table}\left((X,d), \left\{(v,v') \,|v \in \mathcal{D}(X)\,, v' \in \mathcal{D}(d)\,, \sum_{i=1}^{n} \widetilde{\delta}(v, s_i)) \geq v'\right\}\right)\,.$$

Case of $\texttt{multiple\_diversity\_dim}_{\delta_{\mathcal{H}}}$

We have created the $\texttt{multiple\_diversity\_dim}_{\delta_{\mathcal{H}}}$ constraint from the diversity constraint. We remark that it is equivalent to a soft version of the `alldifferent` constraint [1], introduced in [27], which is also equivalent to a soft version of the `allequal` constraint (introduced in [25]).

Definition 41 ($\texttt{soft\_alldifferent}_G^{\min} \equiv \texttt{soft\_allequal}_G^{\max}$ [25, 27])**.**
The $\texttt{soft\_alldifferent}_G^{\min}$ constraint counts the number of inequalities that are not respected in the `alldifferent` constraint, i.e.

$$\texttt{soft\_alldifferent}_G^{\min}(X_1, \ldots, X_n, N) \Leftrightarrow N \geq |\{(i,j) | X_i = X_j, i < j\}|\,.$$

Soft constraints are an important modelling tool. They allow the modelling of over-constrained problems, where some constraint may be violated (i.e. not satisfied). Soft versions of constraints also allow to quantify how far the constraint is to be satisfied. For example in a scheduling problem, workers are legally constrained to work at most a certain amount of time. However, it is possible for some workers to work overtime. This overtime

1. Also called `soft_alldifferent_ctr` in the global constraint catalogue `http://sofdem.github.io/gccat/gccat/Csoft_alldifferent_ctr.html`

can be quantified using soft constraints. Then the overtime can be constrained depending on other laws (for example there may be a maximum amount of overtime allowed).

In a sense, soft constraints allow to measure the *distance* between the fully constrained problem and the solution of the soft problem. We now show the equivalence between the diversity constraint and the soft constraint.

Proposition 7. *Let $X_1, \ldots, X_k$ be k variables. When using the Hamming distance, the* ***multiple_diversity_dim***$_{\delta_{\mathcal{H}}}$ *constraint can be rewritten as a* ***soft_alldifferent***$_G^{\min}$ *constraint:*

$$\textbf{\textit{multiple\_diversity\_dim}}_{\delta_{\mathcal{H}}}(X_1, \ldots, X_k, d) \Leftrightarrow \textbf{\textit{soft\_all\_different}}_G^{\min}\left(X_1, \ldots, X_k, \frac{k(k-1)}{2} - d\right).$$

Proof.

$$\begin{aligned}
&\texttt{multiple_diversity_dim}_{\delta_{\mathcal{H}}}(X_1, \ldots, X_n, d) \\
&\quad \Leftrightarrow \sum_{1 \leq i < j \leq k} \mathbb{1}_{X_i \neq X_j} \geq d \\
&\quad \Leftrightarrow |\{(i,j) | X_i \neq X_j, 1 \leq i < j \leq k\}| \geq d \\
&\quad \Leftrightarrow \frac{k(k-1)}{2} - |\{(i,j) | X_i = X_j, 1 \leq i < j \leq k\}| \geq d \\
&\quad \Leftrightarrow \frac{k(k-1)}{2} - d \geq |\{(i,j) | X_i = X_j, 1 \leq i < j \leq k\}| \\
&\quad \Leftrightarrow \texttt{soft_all_different}_G^{\min}\left(X_1, \ldots, X_k, \frac{k(k-1)}{2} - d\right)
\end{aligned}$$

□

In [27], an algorithm for the arc consistency in $\mathcal{O}(n \cdot m)$ is presented, where n is the number of variables in the constraint and m is the sum of the sizes of the domains. This algorithm can also be used to propagate arc consistency on the $\texttt{multiple\_diversity\_dim}_{\delta_{\mathcal{H}}}$ constraint using Proposition 7.

Case of $\texttt{multiple\_diversity\_dim}_{\delta_{l_1}}$

To our knowledge, the $\texttt{multiple\_diversity\_dim}_{\delta_{l_1}}$ constraint has never been studied as a global constraint. The goal is to find the values of variables that maximise the sum

of all the pairwise distances (absolute value of the difference) between two variables, i.e.

$$\sum_{i=1}^{k}\sum_{j=i+1}^{k}|X_i - X_j| \geq d\,.$$

The first (and naive) way to implement this constraint is to create variables for all the differences $d_{ij} = |X_i - X_j|$ and sum these variables. There will be $k(k-1)/2$ such variables. Splitting the formula between these variables loses the information that each variable occurs in multiple d_{ij}, thus reducing the power of propagation.

The absolute values are the hard part of this sum. They prevent the formula from being factorised. However, if the variables X_i are sorted, we can safely remove the absolute values ($a \geq b \Rightarrow |a-b| = a-b$). The following proposition factorises the formula when the values are sorted.

Proposition 8. *Let $x_1, \ldots, x_k \in \mathbb{Z}$ such that $x_1 \leq x_2 \leq \ldots \leq x_k$. Then*

$$\sum_{i=1}^{k}\sum_{j=i+1}^{k}|x_i - x_j| = \sum_{i=1}^{k}(2i - k - 1)x_i$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{k}\sum_{j=i+1}^{k}|x_i - x_j| &= \sum_{i=1}^{k}\sum_{j=i+1}^{k}(x_j - x_i) \\
&= \sum_{i=1}^{k}(i-k)x_i + \sum_{i=1}^{k}\sum_{j=i+1}^{k}x_j \\
&= \sum_{i=1}^{k}(i-k)x_i + \sum_{j=1}^{k}\sum_{i=1}^{j-1}x_j \\
&= \sum_{i=1}^{k}(i-k)x_i + \sum_{j=1}^{k}(j-1)x_j \\
&= \sum_{i=1}^{k}(2i-k-1)x_i
\end{aligned}$$

□

The only requirement for this formula is that the x_i must be sorted. If we can ensure that the variables in the constraint are sorted, then we can use this formula to reformulate the constraint. To ensure that the variables are sorted, we can use the $\texttt{sort}(\mathcal{X}, \mathcal{Y})$, which

ensures that there is a one-to-one correspondence between the variables in $\mathcal{X}$ and in $\mathcal{Y}$, and that the variables $Y_1, \ldots, Y_n$ are sorted in ascending order. This gives a reformulation of the constraint.

Proposition 9. *We introduce new variables $\mathcal{Y} = \{Y_1, \ldots, Y_k\}$ with domains equal to the union of the domains of the variables in $\mathcal{X}$. We can reformulate the constraint as follows*

$$\texttt{multiple\_diversity\_dim}_{\delta_{l_1}}(X_1, \ldots, X_k, d) \Leftrightarrow \begin{cases} sort(\mathcal{X}, \mathcal{Y}) \\ \sum_{i=1}^{k} (2i - k - 1) Y_i \geq d \end{cases}$$

Proof. The sort constraint ensures that the variables Y_i are sorted, so we can apply Property 8. □

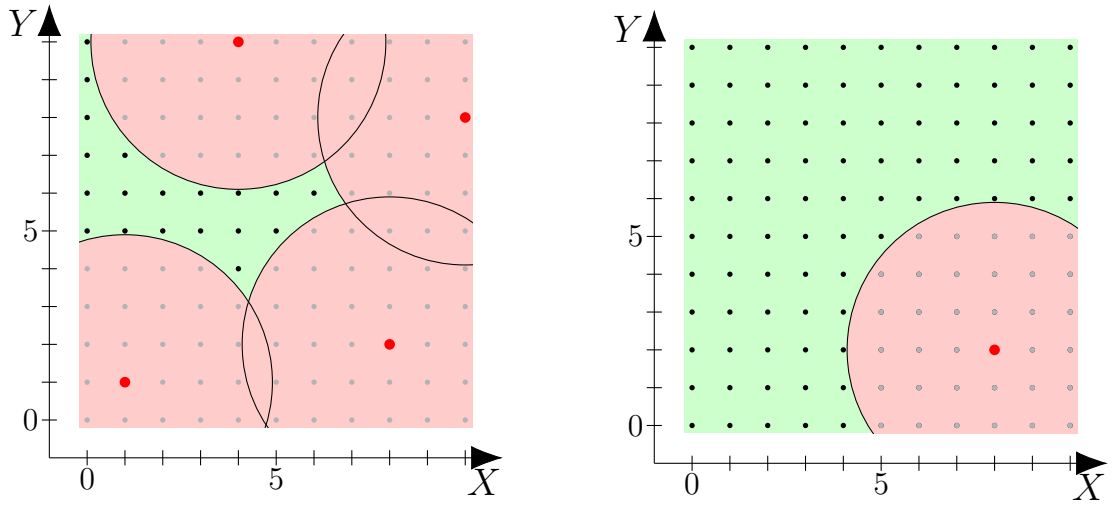
Using this reformulation, a change in the domain of d can be propagated to the variables Y_i and then to the variables X_i. It is also possible to use the fact that the Y_i are sorted when propagating the constraint $\sum_{i=1}^{k}(2i - k - 1)Y_i \geq d$. For example, the constraint `increasing_sum` $(Y_1, \ldots, Y_k, d)$ states that $\sum_i Y_i \geq d$ and the variables Y_i are sorted, and the bound consistency can be done in linear time $\mathcal{O}(k)$ [43]. In our case the sum is weighted, so the consistency algorithm should be adapted. Also, arc-consistency of the `sort` constraint is NP-hard [52], so only an approximation of arc-consistency can be performed for high value of k.

Remark. *If there are no other constraints, maximising the sum $\sum_{i=1}^{k}(2i - k - 1)X_i$ is straightforward. It can simply be done by choosing the smallest possible values for the variables X_i for $1 \leq i \leq \lfloor k/2 \rfloor$ (i.e. for negative coefficients $2i - k - 1$), and the largest possible values for the variables X_i for $\lfloor (k+1)/2 \rfloor < i \leq k$ (i.e. for positive coefficients $2i - k - 1$). We also note that when k is odd, $X_{\lceil k/2 \rceil}$ has a coefficient 0, so has no effect on the distance.*

8.3.2 min Aggregator

The second aggregator we study is the minimum min of the pairwise distances. When using the $\sum$ aggregator, the constraints can be split into smaller constraints on each dimension. This is not possible when using the min aggregator. Propagating arc consistency on the $\texttt{single\_diversity}_{\min,\delta}$ is thus much more complicated.

Theorem 8 ([24])**.** *Arc consistency is NP-hard to propagate on $\texttt{single\_diversity}_{\min,\delta_{\mathcal{H}}}$.*



(a) With $\texttt{single\_diversity}_{\min,\delta_{l_2}}$

(b) With $\delta_{l_2}((X,Y),(10,0)) \geq 6$

Figure 8.2 – Examples of propagation of the diversity constraint using the min aggregator.

Corollary 1. *Arc consistency is NP-hard to propagate on* $\texttt{multiple\_diversity}_{\min,\delta_{\mathcal{H}}}$.

In [24], this result is proved only for the Hamming distance, here we extend it to all the l_p distances (this includes the Manhattan and Euclidean distances).

Theorem 9. *For any* $p \geq 0$*, arc consistency is NP-hard to propagate on* $\texttt{single\_diversity}_{\min,\delta_{l_p}}$ *and on* $\texttt{multiple\_diversity}_{\min,\delta_{l_p}}$.

Proof. In Appendix A.3.1 □

Example. *We use the same problem as in the previous example in Figure 8.1, but with the* min *aggregator and the Euclidean distance. Figure 8.2a shows the solution space (in green) of the* $\texttt{single\_diversity}_{\min,\delta_{l_2}}$*. We see that there is no support for the values* $x \in \{7,\dots,10\}$ *in* $\mathcal{D}(X)$*, nor for the values* $y \in \{0,\dots,3\}$ *in* $\mathcal{D}(Y)$*. This does not mean that these values can be propagated easily by the solver.*

To propagate a min *constraint, the solver reformulates it as multiple inequalities:*

$$\min_{s\in\mathcal{S}} \delta(\mathcal{X},s) \geq d \Leftrightarrow \forall s \in \mathcal{S}, \delta(\mathcal{X},s) \geq d\,.$$

However, this reformulation loses propagation power because fewer values can be propagated. For example, Figure 8.2b shows the solution space of the constraint $\delta_{l_2}((X,Y),(8,2)) \geq$

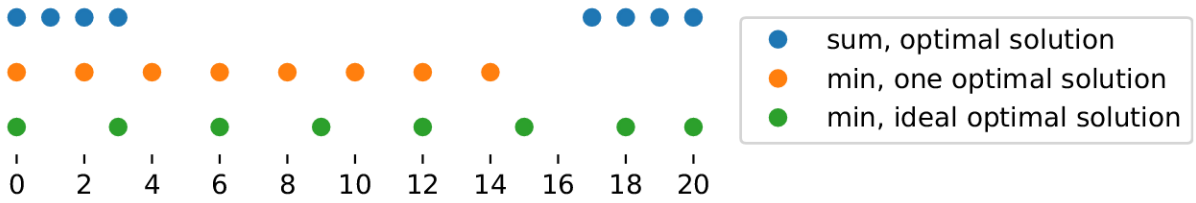


Figure 8.3 – Diversity in one dimension

d. We see that no value of $\mathcal{D}(X)$ *or* $\mathcal{D}(Y)$ *can be propagated because there is always a support.*

The fact that this reformulation loses propagation power is one of the reasons why propagating arc consistency on the $\texttt{single\_diversity}_{\min,\delta_{l_p}}$ *is NP-hard.*

8.4 Graphical Examples

In this section, we show examples of solutions to the `MaxDiverseKSet` problem in few dimensions (up to 3, for graphical presentation).

8.4.1 In One Dimension

In one dimension, we consider the problem with one variable $X \in \{0, \ldots, 20\}$, without constraint, and we search for the most diverse set of 8 solutions. We use the Manhattan distance (equivalent to all the δ_{l_p} for $p > 0$ in one dimension). Figure 8.3 shows three sets of solutions.

The first (in blue) set of solutions is the most diverse set of 8 solutions when using the $\sum$ aggregator. A striking fact is that the solutions are clustered around the ends of the domain (0 and 20). This may seem strange, but when optimising the sum of distances it is better to have some couple of solutions with large distances, even if others are close to each other. Proposition 8 also explains this fact because to optimise the weighted sum, it is better to take extreme values.

The second solution set (in orange), is a most diverse set of solutions using the min aggregator. The minimum distance between solutions is 2 and is achieved between all the solutions. Although this set is optimal, it completely misses a part of the solution space.

When using the min aggregator, an ideal solution set be the third solution set (in green), because it is the one that allows for the largest distances between all the solutions. The minimum distance is 2, and it is only reached between 2 solutions. All the other

pairwise distances are greater than 3.

It would be very difficult to constrain the solver to return the third solution set instead of the second solution set. Ideally, not only the minimum distance should be maximised, but also the second minimum distance, the third, and so on. This means that the most diverse set is the one that maximises the lexicographic order of the sorted pairwise distances. This adds another level of complexity when the set of all possible solutions is unknown. In the following Chapter 9 Section 9.6.1 we show how to optimise the lexicographic order in a reasonable time when the set of solutions is known and is a one-dimensional set (such as a Pareto front in two dimensions).

8.4.2 In Two and Three Dimensions

In two and three dimensions we generate 500 random points in $[0,1]^k$ where $k = 2$ or 3. We use the greedy approach (iteratively finding the point the most distant from the ones previously found) with the Euclidean distance to extract 20 solutions, starting from a randomly chosen one (in blue in the plots).

Figure 8.4 shows the allowed solutions in green and the 20 selected solutions in red, for the two aggregators. It also shows as a heat map the distance from any point to the returned solution set.

When using the min aggregator, the solutions are well distributed in $[0,1]^2$. The heat map shows the regions that are further away from the selected points with a darker shade. This draws the Voronoi diagram of the selected points.

When using the $\sum$ aggregator, the solutions are clustered around the corners of the space. This is similar to the behaviour in one dimension, which could be explained by the reformulation of the $\texttt{multiple\_diversity\_dim}_{\delta_{l_1}}$. This extends to multiple dimensions if separable distances are used. Interestingly, the plot shows that there is the same behaviour with non separable distances, such as the Euclidean distance in our case.

Figure 8.5 shows the same experiment in three dimensions. Using the min aggregator the solutions returned are well distributed in $[0,1]^3$. Using the $\sum$ aggregator, the solutions are again clustered around the corners of the $[0,1]^3$ cube but not in the inside of the cube.

From the point of view of solution space coverage, using the $\sum$ aggregator seems to be a very bad solution. However, we can also see that it finds the boundary of the solution space. In some cases this can be very important, for example when testing corner cases of a software.

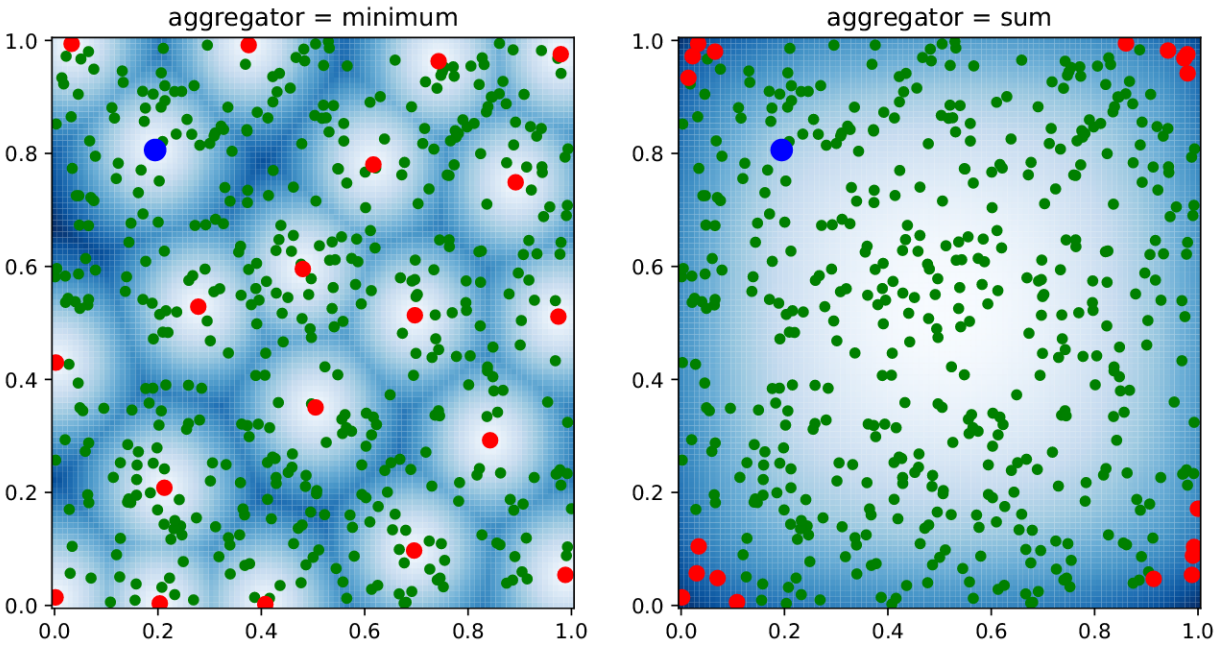


Figure 8.4 – Diversity in 2 dimensions. 500 random points (in green) are generated in $[0, 1]^2$. One starting point is selected at random (in blue), and then 19 points (in red) are selected by the greedy approach, using the min or $\sum$ aggregator and the Euclidean distance. The shade of blue is a heat map of the distance to the solutions selected, i.e. if the heat map is a dark blue, it means that this point is far from the other points selected (using the aggregator).

8.5 Approximation algorithms

Exactly solving `MaxDiverseKSet` is expensive because the model is copied k times. To be able to find solutions faster, approximations can be used.

8.5.1 Deterministic Approaches

We first present deterministic approaches. These two approaches are based on the `MostDistant` problem.

Greedy approach [24] A first approximation is to use the `MostDistant` problem. By iteratively solving `MostDistant` by finding the solution that is most distant from all the previous solutions, the final set of solutions will be an approximation of the `MaxDiverseKSet` problem. This method is called the *greedy* approach. Note that the first solution returned by the greedy approach is not specified. It can be the first solution returned by

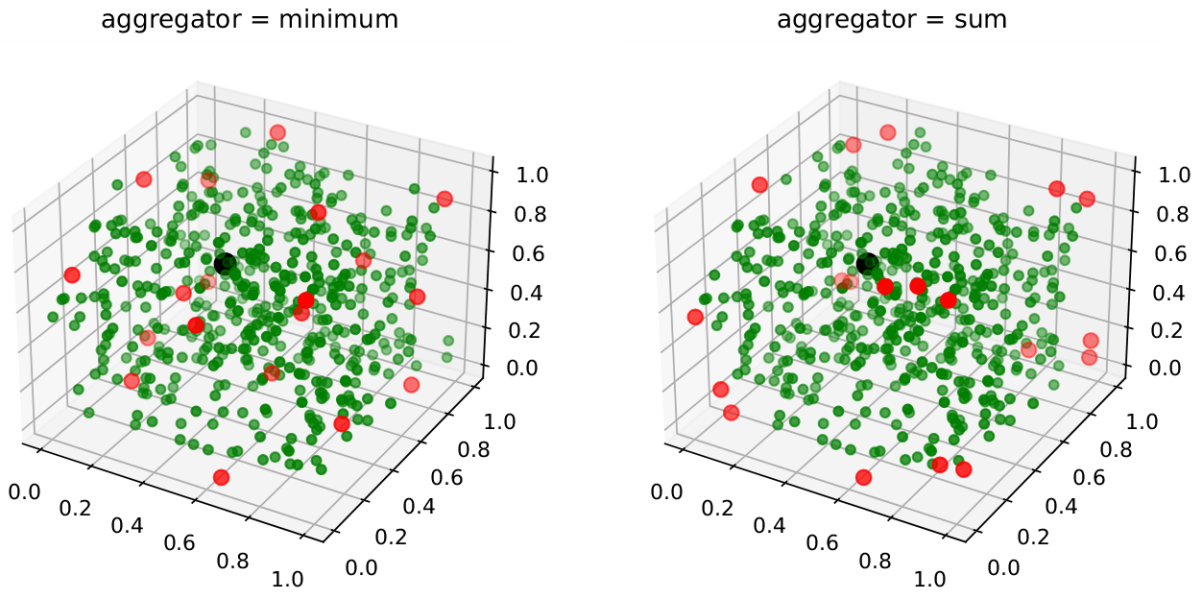


Figure 8.5 – Diversity in 3 dimensions. 500 random points (in green) are generated in $[0, 1]^2$. One starting point is selected (in blue), and then 19 points (in red) are selected by the greedy approach, using the min or $\sum$ aggregator and the Euclidean distance.

a solver, or it can be chosen at random.

Hybrid approach [28] One can also think of an improvement to this greedy approach by first finding k' (with $k' < k$) solutions using the exact approach, and then using the greedy approach to find the remaining $k - k'$ solutions. If k' is small enough, we can hope to solve the exact approach quickly, and then use the greedy approach to quickly generate the remaining solutions. This method is called the *hybrid* approach.

These two approaches trade off quality of diversity for the speed of computation. The following proposition gives a guarantee for the quality of the diversity.

Proposition 10 (1/2-approximation)**.** *When using the* min *aggregator, the greedy and hybrid approaches are 1/2-approximations of the* `MaxDiverseKSet` *problem, i.e. if* $\mathcal{S}_{opt}$ *is the optimal diverse set of size* k*,* d_{opt} *is the minimum pairwise distance of* $\mathcal{S}_{opt}$*,* $\mathcal{S}_g$ *is the set returned by the greedy or hybrid approach, and* d_g *its minimum pairwise distance, then* $\frac{1}{2}d_{opt} \leq d_g \leq d_{opt}$

Proof. In Appendix A.3.2 □

This proposition ensures that in the worst case, the solution set of the greedy or hybrid approach is not too much worse than the optimal solution. The following proposition shows that this worst case is achievable.

Proposition 11. *There exist problems where the minimum distance of the solution returned by the greedy or hybrid approaches is exactly half the minimum distance of an optimal solution.*

Proof. In Appendix A.3.2 □

These propositions show that there is a theoretical guarantee when using a heuristic method, but this guarantee is far from the optimal. Since the exact approach may not be solved to optimality in a reasonable time, it is advantageous to use heuristics because they produce good quality solutions in practice.

Another takeaway from the propositions is the fact that the hybrid approach does not give more guarantee than the greedy approach alone. In fact, finding the k most diverse solutions is not an iterative process: from an optimal set of size k, adding one solution may give a set of size $k+1$ far from the optimal.

8.5.2 Random Approach

We have seen in the previous chapters (in Part III) that randomness can provide good diversity and good coverage. Here, to theoretically analyse the behaviour of random approaches, we focus on the unit square $[0,1]^2$ (or the unit hypercube, if simple formulas can be derived).

$\sum$ Aggregator

We first consider the $\sum$ aggregator. We want to compute the average sum of the distances of k randomly generated points. To do this, we first need to know the average distance between two randomly generated points. This has been studied under the term mean line segment length [187].

Proposition 12 ([187])**.** *Let s_1 and s_2 two random points in $[0,1]^2$, i.e. random variables following the uniform distribution $\mathcal{U}([0,1]^2)$. Then the expected Euclidean distance between s_1 and s_2, noted $\Delta(2)$ is*

$$\Delta(2) = \mathbb{E}\left(\delta_{l_2}\left(s_1, s_2\right)\right) = \frac{2 + \sqrt{2} + 5\ln(1+\sqrt{2})}{15} \approx 0.52\,.$$

Since the expected value is a linear function, we can easily calculate the expected sum of the distances of randomly generated points.

Corollary 2. *Let $\mathcal{S} = \{s_1, \ldots, s_k\}$ be a set of independent and identically distributed points following the uniform distribution $\mathcal{U}([0,1]^2)$ on the unit square. Let Z be the sum of the Euclidean distances, i.e.*

$$Z = \sum_{\substack{s,s' \in \mathcal{S} \\ s \neq s'}} \delta_{l_2}(s, s') ,$$

then

$$\mathbb{E}(Z) = k(k-1)\Delta(2) .$$

Proof.

$$\begin{aligned}
\mathbb{E}(Z) &= \mathbb{E}\left(\sum_{\substack{s,s' \in \mathcal{S} \\ s \neq s'}} \delta_{l_2}(s, s') \right) \\
&= \sum_{\substack{s,s' \in \mathcal{S} \\ s \neq s'}} \mathbb{E}(\delta_{l_2}(s, s')) && \text{(by linearity of } \mathbb{E}) \\
&= \sum_{\substack{s,s' \in \mathcal{S} \\ s \neq s'}} \Delta(2) && \text{(by Proposition 12)} \\
&= k(k-1)\Delta(2)
\end{aligned}$$

□

We can compare this value with the optimal set of points. Using the Manhattan distance, it can be shown (using the Propositions 5 and 9) that the optimal solution set has solutions in the corners of the unit square. In the case where k is a multiple of 4, there are $k/4$ solutions at each corner of the unit square. We assume that the solutions of the same corner differ by a value $\epsilon > 0$, which we neglect (it can be chosen to be as close to 0 as we want). Figure 8.4 and 8.5 show that the same reasoning works for the non separable Euclidean distance. Each solution has a distance 0 from the solutions in the same corner (by neglecting ϵ), a distance $\sqrt{2}$ from the $k/4$ solutions in the opposite

corner, and a distance 1 from the $k/2$ other solutions. The sum of the distances is then

$$\sum_{s \in \mathcal{S}} 1 \cdot \frac{k}{2} + \sqrt{2} \cdot \frac{k}{4} = k^2 \frac{2+\sqrt{2}}{4} \approx 0.85k^2 \,.$$

This means that the random approach is on average 61% worse than the optimal solution set (in the unit square).

In higher dimension The average distance $\Delta(n)$ of two randomly chosen points in the hypercube $[0,1]^n$ is difficult to compute. The one-dimensional case is simple, with $\Delta(1) = 1/3$, and a closed form formula is known for $\Delta(3)$ (also known as the Robbins constant), $\Delta(4)$ and $\Delta(5)$. Only bounds of $\Delta(n)$ are known for higher dimensions, but the values can be computed approximately ($\Delta(n) \approx \sqrt{n/6 - 7/120}$ is a good approximation [161]).

We can compute the optimal sum of distances using solutions at the corners of the hypercube. From a corner, the distance to the $k/2^n$ solutions in the corner with i changes is $\sqrt{i}$, and there are $\binom{n}{i}$ such corners. So the sum of distances is

$$\frac{k^2}{2^n} \sum_{i=0}^{n} \binom{n}{i} \sqrt{i} \,.$$

This allow the ratio between the optimal and the random solution set to be calculated. On average, as the dimension increases, the random solution is 57% worse than the optimal solution.

Manhattan distance Using the Manhattan distance, the same reasoning can be done, but the computations are simpler. The problem can be split into each dimension. On a line $[0,1]$ the average distance between two points is $\Delta(1) = 1/3$. This means that in the hypercube $[0,1]^n$ the average distance between two points is $n/3$, so the average distance between k points is $k(k-1)n/3$.

The sum of the distances between k points at the corners of the hypercube is

$$\frac{k^2}{2^n} \sum_{i=0}^{n} \binom{n}{i} i = \frac{k^2 n}{2} \,.$$

This means that on average, a randomly generated solution set has a sum of distances 66% worse than the optimal set.

min **Aggregator**

Computing the average value of the minimum of the distances of points in the unit square is also difficult.

In one dimension The average minimum distance is known [160] in one dimension (i.e. for random variables in $[0, 1]$).

Proposition 13 ([160])**.** *Let $X_1, \ldots, X_k$ be k independent and identically distributed random variables following the uniform distribution over the unit line $\mathcal{U}([0, 1])$. Let $Z = \sum_{1 \leq i < j \leq k} |X_i - X_j|$ be the minimum distance between the random variables (in one dimension, the Euclidean distance is equivalent to the Manhattan distance). Then $\mathbb{E}(Z) = 1/(k^2 - 1)$.*

The optimal solution set is to space the points evenly, so the minimum distance is $1/(k - 1)$. The average minimum distance is thus smaller than the optimal minimum distance by a factor n, so the approximation factor tends to 0 when k grows.

In higher dimension In higher dimension, the minimum distance of k randomly chosen points may be studied using extreme value theory. For example, the Fisher-Tippett-Gnedenko theorem [170] gives a convergence result for the distribution of the maximum of random variables. However, here we do not know explicitly the distribution of the distance between two random points. However, from experimental simulations, we can make the following conjecture.

Conjecture 1. *Let $X_1, \ldots, X_k$ be k independent and identically distributed random variables following the uniform distribution over the unit hypercube of dimension n $\mathcal{U}([0, 1]^n)$. Let $Z = \sum_{1 \leq i < j \leq k} \delta_{l_2}(X_i, X_j)$ the minimum distance between the random variables. Then, there is a constant c such that $\mathbb{E}(Z) \sim c/k^{2/n}$.*

We can easily give a lower bound on the optimal minimum distance of a solution set. When can place the solutions in a lattice (n-dimension grid) where the minimum distance is $\lfloor 1/k^{1/n} \rfloor$. The minimum distance in the order of $1/k^{1/n}$ of the optimal solution should be compared with the expected minimum distance of a random solution set $(1/k^{2/n})$. This means that the approximation factor of the random sampling also tends to 0 as n grows.

Conclusion The analysis of random sampling shows that optimising the minimum distance is harder than optimising the sum of the distances. To optimise the minimum distance, solutions should be well chosen in the solution space, and this task is difficult. Random sampling cannot give a guarantee of approximation.

When optimising the sum of the distances, it can be expected that random sampling will give a decent solution set. On average, the sum of the distances of the solution set returned by random sampling is close to the optimal one, up to a constant factor.

8.6 Conclusion

In this chapter, we returned to the definitions of the diversity problems presented in [24]. To solve these problems, we defined diversity constraints. We studied these constraints depending on the aggregator ($\sum$ or min) and the distance used. We were able to reformulate the diversity constraints using the $\sum$ aggregator into smaller constraints, we showed a propagation algorithm, and a link to soft constraints. We experimentally analysed the solution sets returned by using either the $\sum$ or the min aggregator. The solution sets have specific characteristics, with solutions in different parts of the search space. We also analysed the quality of the approximation algorithms (greedy, hybrid, and random) and showed approximation factors that ensure the quality of the returned solution set compared to the optimal solution set.

This chapter bridges the gap between three domains:

- At the start of the chain are the users. They want to be presented with solutions to have an understanding of their possibilities. Depending on their needs, the solutions should either cover the solution space well (using the min aggregator), or find extreme points (using the $\sum$ aggregator).
- Between the solver and the user are the modellers. Their role is to translate the user's needs into a constraint programming model. It is also their role to know what degree of approximation is allowed. We give to the modellers theoretical guarantees about the behaviour of the approximation algorithms.
- At the end of the chain, solver developers have to implement the diversity constraints used by the modeller. For these developers, we have shown how the diversity constraints can be reformulated (without loss of propagation power) as smaller constraints, or already existing constraints. This facilitates the development phase by reusing basic blocks.

In this chapter, we have assumed that the problem is specified as a satisfaction problem. However, in many cases multiple (possibly conflicting) objectives are to be optimised. The following chapter studies diversity in a multi-objective framework and shows how to define the diversity problems and constraints in this framework.

9		3				5		8
5	7		1		8		6	3
4			3		5			9
			2		7			
8	3						2	7
			6		9			
		8		4		3		
	9						5	

Figure 8.6 – A *Sudoku* grid.

Chapter 9

Application: Diversity in a Multi-objective Problem

This chapter describes an ongoing collaboration with members of the R-IOSuite project. In this project, the end-user is a prefect making decisions that have impact on the life of people.

9.1 Introduction

In the previous chapters, we developed methods that are studied theoretically. But the only way to evaluate these methods is practical, by presenting solutions to a user. Hence, we started a collaboration with the RIO-Suite project in order to see how we could adapt our work to a real-life problem. Here the end-user is a decision maker in a high stakes situation (monument fire, natural disaster, etc). This decision maker must choose between multiple objectives to optimise: the duration of the intervention, the cost, the expertise of the agents, etc.

In a multi-objective problem, even if the solutions are well defined by constraints, it is more difficult to find an *optimal* solution. The objectives may be conflicting, so the solutions cannot be compared. In satisfaction problems, the diversity has been defined between the instantiations of the solutions (the actual values of the variables). In multi-objective problems, however, solutions are first compared by their objective values. The users must understand how the objectives interact, and what the possible values of the objectives are before making a decision. This way, the diversity is not defined on the solutions, but rather in the objective space.

In this chapter, we propose an approach inspired by PostHoc [28] to find a good set of diverse solutions to present to a user. This approach first finds solutions to the problem, and then extracts a subset of diverse solutions from them.

We first present an application in Section 9.2. This application shows the challenges of

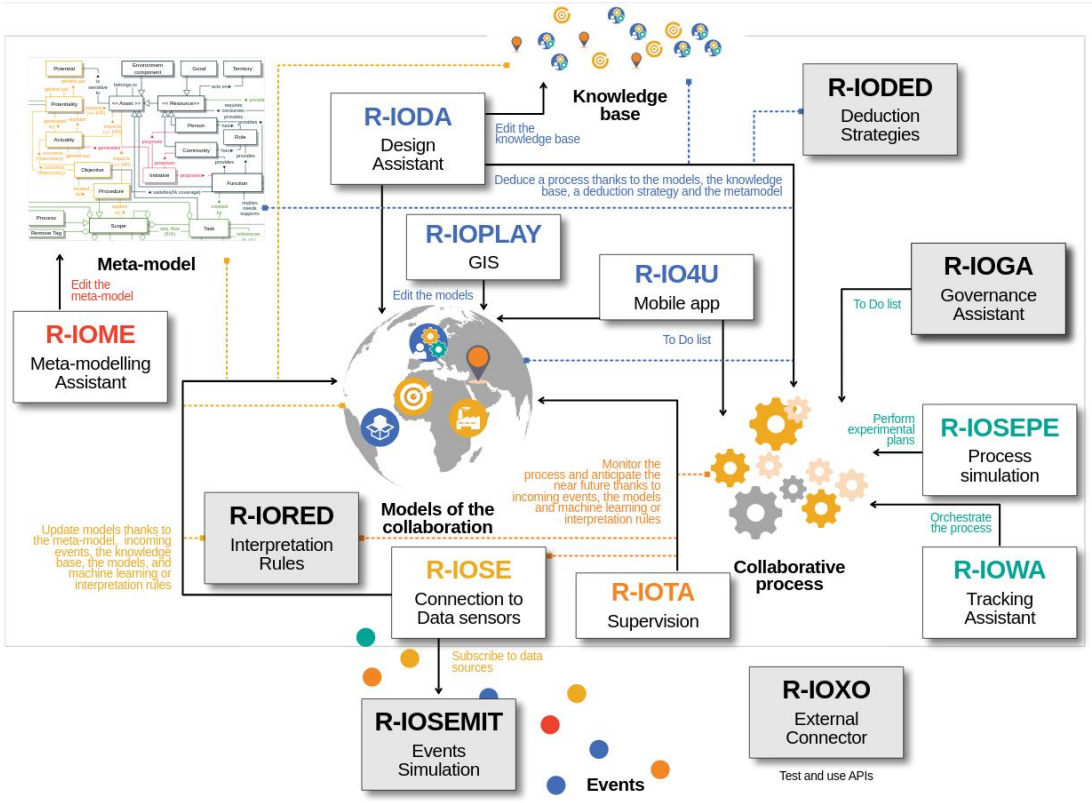


Figure 9.1 – Component diagram of the R-IOSuite software suite.

finding solutions to multi-objective problems, and interacting with a user. In Section 9.3 we then formally define the multi-objective framework. In Section 9.4 we present algorithms to generate diverse weights in the simplex, a prerequisite for most of the following approaches. We then present algorithms to perform the two steps of the approach we use in Sections 9.5 and 9.6.

9.2 R-IO: a Model of Crisis Management

In this section I present an application of diversity in a multi-objective setting. It is based on my preliminary work on the R-IOSuite project. "R-IOSuite is a software suite that embeds a set of tools dedicated to support efficiently inter-organizational collaborations (collaborative industrial projects, supply chain, crisis management, etc)". [1] R-IOSuite is a large project with several components. Figure 9.1 shows the different components of the R-IOSuite project. It includes tools for modelling the problem, a mobile application,

1. `https://r-iosuite.atlassian.net/wiki/spaces/RIOSUITE/overview?mode=global`

tools to simulate events, etc. Of all these components, we focus on the R-IODA component: the design assistant. This is the component that translates the domain knowledge (rules, possible tasks) and the input (agents available, skills) into a constraint satisfaction problem, solves it and proposes the solutions to the users.

9.2.1 Context

Albi's Cathedral [2]

We use an example application to show the purpose of R-IODA (the design assistant of R-IOSuite): the cathedral of Albi (a World Heritage Site) is on fire. The prefect (this can apply to other decision makers, such as mayors or presidents) has to plan the rescue team and the firefighters to save the visitors and stop the fire.

The prefect has a number of agents divided into different classes. In this example, we focus on three classes: firefighters, police officers, and paramedics/doctors. Each class has a number of agents that can be deployed to solve the problem at hand. Each class of agents has a set of skills for specific tasks, for example, paramedics are experts at treating injured people, but firefighters are also trained in first-aid. Depending on the the number of agents available, some firefighters can be assigned to the help of injured people instead of stopping the fire.

There are several tasks that need to be carried out in order to stop the fire safely. Before the fire can be tackled, the injured should be evacuated from the cathedral. Ideally, this would be done by paramedics, but often firefighters are the first to arrive at a scene. Once all the survivors have been evacuated, and when all the preparations have been made (preparing the fire engines and the water), the firefighters can stop the fire. At the same time, a safety zone should be set up around the cathedral to prevent people from entering a dangerous place. This task can be carried out by police officers, a class of agents trained to set up safety zones. At the end of the fire, the structural integrity of the cathedral should be checked before ending the intervention, to prevent, for example, some parts of the cathedral from falling on nearby dwellings. Figure 9.2 graphically represent the dependencies between the tasks. Evacuation and the preparations are prerequisites for

2. By Krzysztof Golik — Own work, CC BY-SA 4.0, `https://commons.wikimedia.org/w/index.php?curid=62150141`

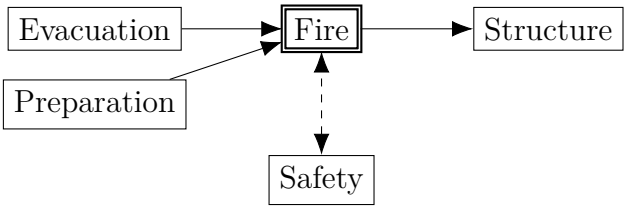


Figure 9.2 – Dependency graph of the tasks to stop the fire. A vertex is a task and an arc is a dependency. The dotted double arc is a support dependency (the two tasks should be done at the same time.

stopping the fire. The safety task (keeping the zone safe) is a support task, i.e. it should be done at the same time as stopping the fire. Then, there is a successor task, to ensure the stability of the structure.

This example is a scheduling problem: agents with different skills should be assigned to perform tasks in a given order. In this example, the total time taken by the schedule (i.e. the time taken to complete the last task) should be as short as possible. This is one of the objectives of the optimisation. However, there are several other objectives, some of which are easier to model than others. We now give some examples of such objectives that the user might want to optimise.

- The expertise of agents assigned to tasks should be maximised: the most skilled agents for a given task should be assigned to that task. However, this may mean delaying tasks in order to always assign the most skilled agent (who may be in another task at the same time).
- The cost should be minimised. In our example, the prefect can ask for help from the fire brigades of other neighbouring towns, but this has a cost (transporting people and equipment).
- The probability of success should be maximised. A model is a partial representation of the reality, and other non-modelled factors may cause a task to fail (for example, failing to stop the fire because of wind). This probability may depend on the expertise of the agents assigned to a task, or the preparation time, or the number of agents assigned.
- The fatigue of agents should be minimised. Depending on the duration of the event, some agents may need to take breaks. For example, during extreme forest fires (which become more and more frequent during the summer), firefighters risk their lives on a daily basis in a stressful environment. Fatigue should be taken into account at the planning stage to protect their mental and physical health.

- The number of lives saved should be maximised. However, it can be very difficult to model this objective with incomplete information. For example, in an earthquake this can be used to prioritise which collapsed buildings should be dug out first (for example, residential areas can be prioritised).
- The *heritage value* of the monument (after all the tasks have been completed) should be maximised. For example, the firefighters might try to stop the fire in the part of the cathedral that contains the valuable paintings first.
- The safety of the agents should be maximised. Taking non-experts agents in a dangerous place may put them at risk.

Some of these objectives (such as costs or the expertise) are easy to model. However, some other objectives may be more difficult to model due to lack of information or modelling approximation. Also, most of these objectives are conflicting:

- The solution that minimises the makespan (total time) is likely to be costly,as a lot of agents, vehicles and equipment will be borrowed from neighbouring cities.
- To save the most people, some low-expertise agents may be assigned to the rescue (in addition to the expert agents), which may also put them at risk.
- Trying to save some parts of the monument may allow the fire to spread further, reducing the likelihood of effectively stopping the fire.

In this multi-objective situation, it is not possible to find *the best* solution. Even if the problem were perfectly modelled, it is not possible to leave the decision to the algorithm alone. The prefect who makes the final decision takes *responsibility* for her/his choices. These decisions have a real impact on whether or not lives are saved. It is very difficult to simplify some of the objectives, as it would require being able to compare them: this would mean giving a price for a life saved, or comparing the safety of the agents to the probability of success. In this setting, it is necessary to give the user (the prefect) a small, well-chosen (i.e. diverse) set of solutions from which to choose. The prefect has to perform the final decision.

9.2.2 CP Model

First we present the data of the problem. There is:

- a set $\mathcal{S}$ of skills;
- a set of tasks to perform $\mathcal{T}$. Each task $t \in \mathcal{T}$ has
  — a duration δ_t;
  — a set of predecessor tasks ρ_t that must be completed *before* t starts;

  - — a set of support tasks σ_t that have to be performed *while* t is performed;
  - — a required skill χ_t;
  - — a consumption $cons_t$.
- a set of agent classes $\mathcal{A}$. Agent classes represent a class of agents (for example, firefighters or policemen). Each agent class $a \in \mathcal{A}$ has
  - — for each skill $s \in \mathcal{S}$ a level $\lambda_{a,s}$;
  - — a capacity $capa_a$, giving the number of agents in the class.

We now define the variables used in the model. For all agent classes $a \in \mathcal{A}$ and tasks $t \in \mathcal{T}$:

- d_t^{start} and d_t^{end} are the starting and ending time of task t;
- $makespan$ is the ending time of the latest task;
- $to_{a,t}$ is a Boolean variable equal to 1 iff the agent class a is assigned to the task t;
- $agent_t$ is the agent class assigned to task t;
- $level_t$ is the skill level of the agent class assigned to task t;
- $total\_skill$ is the sum of all skills of the agent classes at their assigned tasks.

We now give the constraints of the model. First we link the variables $to_{a,t}$ to $agent_t$.

$$\forall t \in \mathcal{T}, agent_t = a \Leftrightarrow to_{a,t} = 1\,.$$

This also ensures that all tasks are performed by a single agent class (enforced by the redundant constraint $\forall t \in \mathcal{T}, \sum_{a\in\mathcal{A}} to_{a,t} = 1$). We need to ensure that at all time, the consumption of the tasks assigned to an agent class is not higher than the capacity of this class. This is exactly represented by a `cumulative` constraint.

$$\forall a \in \mathcal{A}, \texttt{cumulative}\left(\{(d_t^{start}, d_t^{end}, cons_t \times to_{a,t}) \mid t \in \mathcal{T}\}, capa_a\right)\,.$$

The `cumulative` constraint takes as parameter a set of triples containing a starting time, an ending time, and a consumption, and ensures that the sum of the consumption of overlapping tasks does not exceed the capacity. In our case, the consumption is either $cons_t$ if the task is assigned to the agent class ($to_{a,t} = 1$), or 0 otherwise. We first constrain the starting and ending times of the tasks. All the predecessor tasks must be completed before the current task.

$$\forall t \in \mathcal{T}, \forall t' \in \rho_t, d_{t'}^{end} \leq d_t^{start}\,.$$

All the support tasks must be performed at least at the same time as the considered task.

$$\forall t \in \mathcal{T}, \forall t' \in \sigma_t, d_{t'}^{start} \leq d_t^{start} \wedge d_t^{end} \leq d_{t'}^{end} .$$

The makespan is the ending time of the latest task, i.e. the maximum ending time.

$$makespan = \max_{t \in \mathcal{T}} d_t^{end} .$$

The skill level of the agent class assigned to the the task is constrained by the following constraints.

$$\begin{aligned} &\forall t \in \mathcal{T}, level_t > 0 \\ &\forall t \in \mathcal{T}, \texttt{element}\left(level_t, [\lambda_{a_1,\chi_t}, \ldots, \lambda_{a_{|\mathcal{A}|},\chi_t}], agent_t\right) \\ &\qquad total\_skill = \sum_{t \in \mathcal{T}} level_t \end{aligned}$$

The $\texttt{element}(v, T, i)$ is equivalent to $v = T[i]$. Here we create an array containing the skill level of each agent for the skill χ_t (the skill required for the task t). This `element` constraint forces the $level_t$ to be equal to the level of the agent $agent_t$ on the task χ_t.

9.2.3 Visualisations

When working with a non-expert user, the visualisations are almost as important as the quality of the solutions. Depending on the application, different visualisations are possible, for example a Gantt chart for planning problems, or a plot of the routes for vehicle deliveries. However, in a multi-objective setting, it is also difficult to show the user the quality of the solution. The solution is evaluated against multiple objectives, which may be conflicting. We show here an example of a visualisation to show to the user the differences between the objective values of the solutions.

As an example, we have generated a planning problem with the constraints presented in the previous section. We generated 15 tasks with some dependencies between them, and a random duration (between 10 and 20 minutes). We defined three agents (three classes containing a single agent): their skills for each task were randomly generated (between 0 and 100), and we also randomly generated values for a third dimension (between 0 and 100). We suppose that this third dimension models the safety of the agents, to be maximised (it could be used to model other objectives as well). We try to minimise the

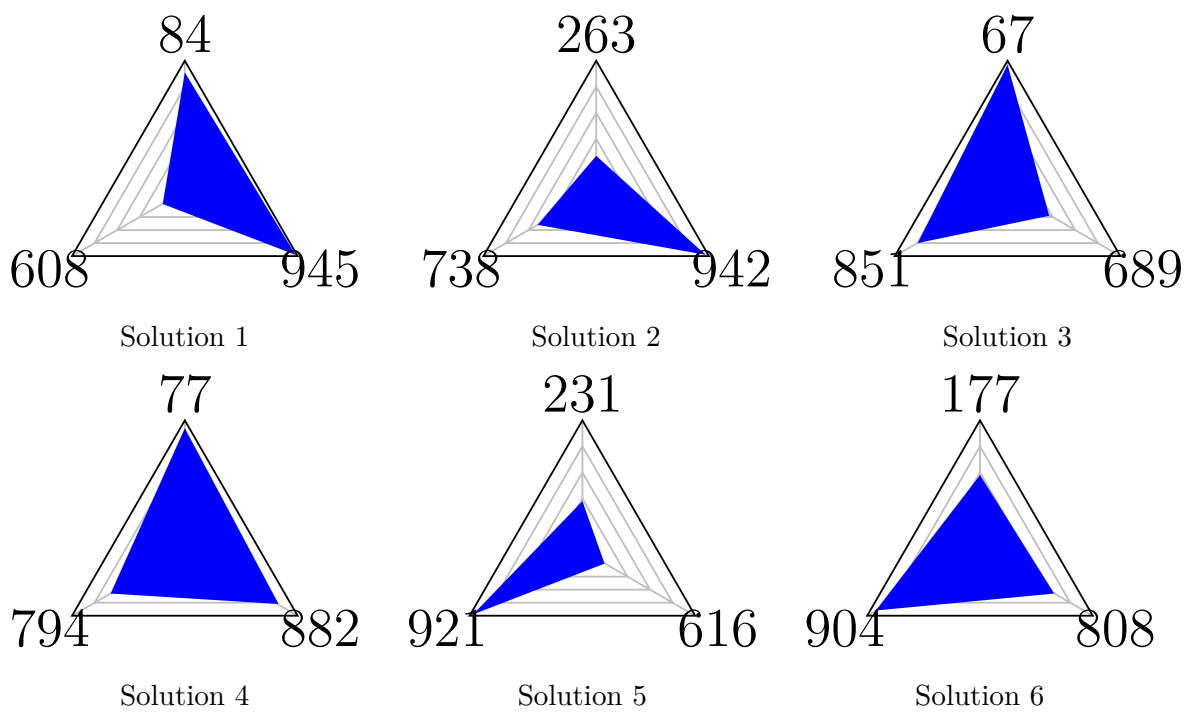


Figure 9.3 – Representation of the three objectives for six solutions. The topmost vertex of the triangles is the value for the makespan, the leftmost is for the skill, and the rightmost is for the third dimension. A big triangle (i.e. with vertices close to the maximum of the spider plot) represents a good solution.

makespan of the problem (i.e. the total time it takes to complete all the tasks), maximise the expertise of the agents (the skill levels), and maximise the safety.

In Figure 9.3 we show the objective values for 6 diverse solutions on spider plots. For example, the first solution has a makespan of 84, a total skill level of 608 and a safety value of 945. The six solutions have very different objective values. For example, the first solution is very good for the makespan and the safety (represented by the blue triangle pointing to the topmost and rightmost vertices of the plot), but the fifth solution is almost the opposite: it has a very good objective value for the total skill, but not for the other two objectives. The six solutions we present here have diverse objective values.

For a user, seeing a representation of the solutions like this can help with the decision. A common way to find *good* solutions for multi-objective problems is to transform the multiple objectives into a single objective by applying a weighted sum. However, it can be difficult for users to choose good weights. Seeing the objective values on some solutions can help them to find weights, or at least express preferences. For example, in Figure 9.3

the fourth solution is good on all objectives, but the fifth solution is almost optimal on the total skills, but very bad on the other two objectives. Seeing these two solutions, a user might say that the improvement in the total skill is not worth having an objective so bad on the other objectives. This kind of visualisation can be used in an interactive setting where the user is shown solutions, and the algorithm learns preferences.

9.3 Definitions

We now formally define the multi-objective optimisation framework.

9.3.1 Multi-Objective Optimisation

In Chapter 2 we defined Constraint Satisfaction Problems (CSPs) and Constraint Optimisation Problems (COPs). We define here COPs with multiple objectives.

Definition 42 (Multi-objective COP)**.** A *multi-objective* constraint optimisation problem is a quadruplet $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \{obj_1, \ldots, obj_m\} \rangle$ where $\mathcal{X}, \mathcal{D}$, and $\mathcal{C}$ are the variables, the domains, and the constraints of the problem, and obj_i are objective variables that should be optimised.

We call *objective space* the projection of the search space on the objective values. We define the function F_{obj} as the mapping between a solution and the objective space, i.e. let $\sigma \in Sols(\mathcal{P})$, then

$$F_{obj}(\sigma) = (\sigma(obj_1), \ldots, \sigma(obj_m)).$$

We say that $F_{obj}\sigma$ is the *objective vector* of σ. We also extend the definition of F_{obj} to sets, i.e. with $\mathcal{S} \in Sols(\mathcal{P})$, then $F_{obj}(\mathcal{S}) = \{F_{obj}(\sigma) \,|\sigma \in \mathcal{S}\}$.

Remark. *In this chapter we assume, without loss of generality, that all objectives should be maximised. If an objective were to be minimised, it is possible to create the variable* $obj' = -obj$ *that should be maximised.*

In the usual COP, the goal is to find a solution that maximises the objective. In multi-objective optimisation, however, the solutions may not be comparable.

Example. *Consider a company that wants to maximise its profits. They also want to maximise the welfare of their employees, which can be modelled as the number of holidays they have. These two objectives are contradictory, there is no single solution that maximises both objectives at the same time. To maximise the profit, the company should not*

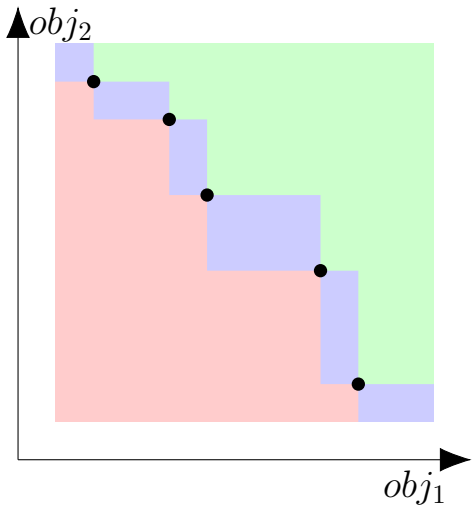


Figure 9.4 – Graphical representation of the dominated, and non-dominated solutions. The dots are the points in $\mathcal{S} = \{(1,9),(3,8),(4,6),(7,4),(8,1)\}$, the area in red are the dominated solutions, in green the dominant solutions, and in blue the non-dominated and non-dominant solutions.

give any holidays to the employees. To maximise the employees happiness, the company can give lots of holidays days, but the profit will be lower.

To define what can be considered as optimal solutions, we need to define an ordering on the objective space.

Definition 43 (Dominance)**.** Let y^1 and y^2 be two objective vectors in the objective space $\mathbb{R}^m$, we define the following orders:

- $y^1 \leq y^2$ (y^2 weakly dominates y^1) iff $\forall i\{1,\ldots,m\}, y^1_i \leq y^2_i$
- $y^1 < y^2$ (y^2 dominates y^1) iff $y^1 \leq y^2$ and $y^1 \neq y^2$

Given two solutions s^1 and s^2 to a problem $\mathcal{P}$, we say that s^2 dominates s^1 (noted $s^1 \prec s^2$) iff $F_{obj}(s^1) < F_{obj}(s^2)$. We note $s^1 \not\prec s^2$ when s^1 is not dominated by s^2.

A set $\mathcal{S} = \{s^1,\ldots,s^k\}$ is said to be non-dominated if $\forall s^i, s^j \in \mathcal{S}, s^i \not\prec s^j$.

This order is a strict partial order, because it is irreflexive, antisymetric, transitive, but not total (i.e. it is false that for all $s \neq s', s \prec s'$ or $s' \prec s$). This means that some solutions cannot be compared.

Example. *In Figure 9.4 shows an example of 5 solutions* $\mathcal{S} = \{(1,9),(3,8),(4,6),(7,4),(8,1)\}$ *in the objective space in two dimensions. This set is a non-dominated set of solutions.* $\mathcal{S}$ *divides the space into three sub-spaces:*

- *The red area represents all the objective vectors dominated by at least one solution in $\mathcal{S}$.*
- *The green area contains all the objective vectors that dominate at least one solution of $\mathcal{S}$.*
- *The blue area contains all the objective vectors that are not dominated by any solution in $\mathcal{S}$.*

Non-dominated sets of solutions contain solutions that cannot be compared. Of all the possible non-dominated sets of solutions, there is an optimal one, called the Pareto set.

Definition 44 (Pareto set/front)**.** Let $\mathcal{P}$ be a multi-objective optimisation problem. The *Pareto set* of $\mathcal{P}$ is the set of all solutions that are not dominated by any other solution, i.e.

$$\{s \in Sols(\mathcal{P}) \mid \forall s' \in Sols(\mathcal{P}), s \not\prec s'\}.$$

The *Pareto front* is the projection of the Pareto set onto the objective space.

An optimal solution to a multi-objective constraint optimisation problem is a solution of the Pareto set.

Ideally, solving a multi-objective optimisation problem boils down to finding the Pareto set. Unfortunately, this set can be exponential in the number of objectives.

9.3.2 Pareto Constraint

Single objective constraint optimisation problems (with a variable *obj* to maximise) are solved by transforming the problem into a satisfaction problem by constraining the objective function. Each time a solution σ (with objective value $\sigma(obj)$) is found, a new constraint $obj > \sigma(obj)$ is added, forcing the solver to find a better solution. When the problem is unsatisfiable, the best solution has been found.

A similar approach is used in multi-objective optimisation. Given a set of non-dominated solutions, a constraint is added forcing the next solution to be non-dominated.

Definition 45 (`pareto` constraint [156])**.** Let $\mathcal{P}$ be a multi-objective optimisation problem, and let $\mathcal{S}$ be a non-dominated set of solutions of $\mathcal{P}$. The `pareto` constraint forces the next solution not to be dominated by any solution in $\mathcal{S}$, i.e.

$$\texttt{pareto}(obj_1, \ldots, obj_m, \mathcal{S}) \Leftrightarrow \forall \sigma \in \mathcal{S}, (obj_1, \ldots obj_m) \not\prec s.$$

1 **Function** *ParetoSolve*($\mathcal{P}$)
  Data: A multi-objective COP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \{obj_1, \ldots, obj_m\} \rangle$
  Result: The Pareto set of $\mathcal{P}$
2 $\mathcal{S} \leftarrow \emptyset$
3 **while** $Sols\left(\mathcal{P} \wedge \texttt{pareto}(obj_1, \ldots, obj_m, \mathcal{S})\right) \neq \emptyset$ **do**
4 $s_{new} \leftarrow$ one solution of $\mathcal{P} \wedge \texttt{pareto}(obj_1, \ldots, obj_m, \mathcal{S})$
5 $\mathcal{S} = \{s \in \mathcal{S} | s \not\prec s_{new}\} \cup \{s_{new}\}$
6 **return** $\mathcal{S}$

Algorithm 9.1: Multi-objective Pareto set solving.

This constraint is called `pareto` because it is used to find the Pareto set. However, it does not directly constrain solutions to be in the Pareto set. It only constrains solutions to be non-dominated by the solutions in $\mathcal{S}$. To find the full Pareto front, an iterative process must be performed, as in single objective optimisation.

Algorithm 9.1 shows the iterative solving process to generate the Pareto set. It maintains a set $\mathcal{S}$ of non-dominated solutions, which is improved until it is equal to the Pareto set. To do this, the problem is solved by constraining solutions to be non-dominated by the solutions in $\mathcal{S}$. When a solution is found, the set $\mathcal{S}$ is updated by removing the solutions dominated by the new solution s_{new} (in line 5). This iterative process stops when there is no solution to the problem with the `pareto` constraint. This ensures that the set $\mathcal{S}$ is the Pareto set.

Algorithm 9.1 generates the Pareto set. However, if the running time is bounded, the set $\mathcal{S}$ can still be returned as an approximation of the Pareto set.

9.3.3 Solution Set Evaluation

We present how to evaluate if a set of non-dominated solutions is *good*, or how to compare two sets of solutions. We present the hypervolume indicator. For detailed surveys about performance indicators we refer the reader to [144, 154, 159]. The hypervolume indicator measures the size of the space dominated by the solutions in the evaluated set.

Definition 46 (Hypervolume Indicator)**.** We extend the notion of closed interval to high dimension. Let l and u be two vectors of $\mathbb{R}^m$, then $[l, u]$ is the set of points that dominate l but are dominated by u, i.e.

$$[l, u] = \{v \in \mathbb{R}^m \mid l \leq v \leq u\} = \prod_{i=1}^{m} [l_i, u_i] \,.$$

Let $\mathcal{S}$ be a set of objective vectors, and let r be a reference point dominated by each vector in $\mathcal{S}$. The hypervolume indicator $\mathcal{H}yp(\mathcal{S})$ is the size of the set of points that dominate r and are dominated by at least one point of $\mathcal{S}$, i.e.

$$\mathcal{H}yp(\mathcal{S}) = \left| \bigcup_{s \in \mathcal{S}} [r, s] \right| .$$

Example. *We use the same example as in Figure 9.4 with the set* $\mathcal{S} = \{(1,9), (3,8), (4,6), (7,4), (8,1)\}$. *We take the reference point* $r = (0,0)$. *The hypervolume in two dimensions is a surface, here the surface in red. In two dimensions it is easy to compute the hypervolume, and in this example* $\mathcal{H}yp(\mathcal{S}) = 44$.

In [157] the authors propose to evaluate a subset by computing the representativeness of the solutions. Each solution in the solution space should be represented by a nearby selected solution.

Definition 47 (Representative Solutions)**.** Let $\mathcal{F}$ be the Pareto front (or an approximation of it), and $\mathcal{S} \subset \mathcal{F}$ a subset of selected solutions. The *radius* of $\mathcal{S}$ is defined as

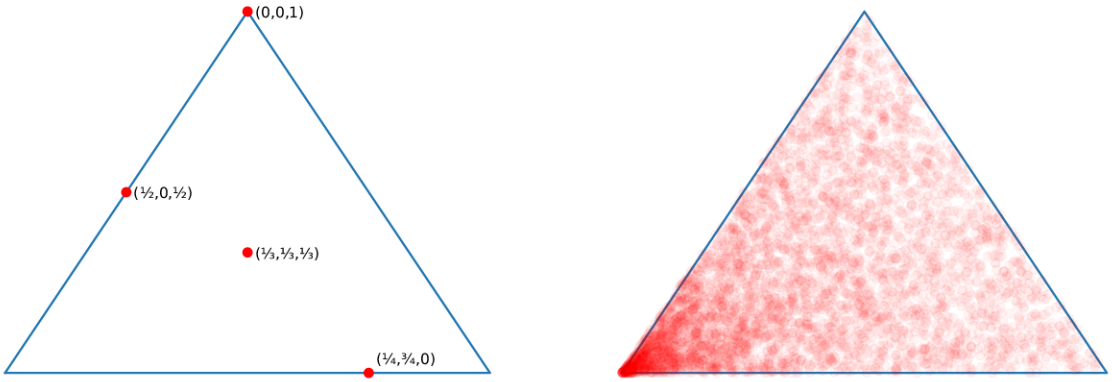
$$\Omega(\mathcal{S}) = \max_{p' \in \mathcal{F}} \min_{p \in \mathcal{S}} \delta(p, p') .$$

In this definition, $\min_{p \in \mathcal{S}} \delta(p, p')$ is the distance between p' and its closest selected solution. $\Omega(\mathcal{S})$ is then the largest distance between one solution and its closest selected solution. To maximise the representativeness of $\mathcal{S}$, the radius should be minimised (i.e. the solutions are represented by nearby selected solutions).

Finally, it is also possible to evaluate the quality of the solution set by considering the minimum pairwise distance, as done in the previous chapter.

9.4 Diverse Weight Generation

In the following sections, we need to generate diverse weights in m dimensions such that the sum of the values is 1. In this section, we make a detour from multi-objective optimisation to present algorithms for generating diverse weights. The sum of the values of the weights we want to generate is equal to 1. In $m + 1$ dimensions, the set of such weights is called the m-simplex.



(a) Example of representation of points in the 2-simplex.

(b) Non-uniform generation of points of the 2-simplex.

Figure 9.5 – Simplex representation

Definition 48 (m-simplex)**.** Let m be an integer (greater than 1). The *standard m-simplex* is defined as

$$\mathcal{W}_m = \left\{ w \in [0,1]^{m+1} \middle| \sum_{i=0}^{m} w_i = 1 \right\} .$$

The m-simplex can be represented by a m-dimensional triangle (a triangle in two dimensions, a tetrahedron in three dimensions) by choosing $m+1$ affinely independent points $u_0, \ldots, u_m \in \mathbb{R}^m$, so that each $w \in \mathcal{W}_m$ is represented by the point

$$\sum_{i=0}^{m} w_i u_i \, .$$

Example. *We consider the 2-simplex $\mathcal{W}_2$. It contains all the vectors $w \in [0,1]^3$ such that $w_1 + w_2 + w_3 = 1$. A vector of $\mathcal{W}_2$ can be represented in two dimensions by choosing three points (the vertices of the triangle). In Figure 9.5a we have chosen three vertices of an equilateral triangle, and plot four vectors of $\mathcal{W}_2$.*

Weights in the simplex are the cornerstone of very different approaches presented in this chapter. In the following sections, they are used as coefficients in a weighted sum (in Section 9.5.2), as a probability distribution (in Section 9.5.3) and as line support vectors (in Section 9.6.2). In all these uses, the weights must be diverse.

Generating diverse weights in the simplex is not an easy task. In [157] the authors state that *"this can be done analytically, or by finely discretising the set $\mathcal{W}_m$ and take advantage of the exact procedure"* `MaxDiverseKSet`. However, they do not give this ana-

1 **Function** *SimplexSample*(m)
  Data: An integer m
  Result: A random (uniformly sampled) vector of $\mathcal{W}_m$
2 **for** $i \in \{0, \ldots, m\}$ **do**
3 $w_i \leftarrow -\ln(1 - \text{Random}())$
4 $s \leftarrow \sum_{i=0}^{m} w_i$
5 **return** $\left(\frac{w_0}{s}, \frac{w_1}{s}, \ldots, \frac{w_m}{s}\right)$

Algorithm 9.2: Uniform sampling in $\mathcal{W}_m$.

lytical procedure. Also, by using a discretisation, and solving exactly the `MaxDiverseKSet` problem, it is only possible to generate very small sets (less than 10 weights) because of the combinatorial explosion of the cases the solver has to search.

We now present three different ways to generate diverse simplex weights. A survey of simplex point generation is done in [149] with more algorithms than the ones presented here.

9.4.1 Random Generation

A first way to generate (approximately) diverse weights from $\mathcal{W}_m$ is to pick them randomly. However, this random generation should be done with care if a uniform distribution is desired. The naive way to generate a random weight is to pick the value w_0 of the first dimension uniformly in $[0, 1]$, and then the next value in $[0, 1 - w_0]$, the next value in $[0, 1 - w_0 - w_1]$, etc., and the last dimension is fixed to $1 - \sum_{i=0}^{m-1} w_i$. However, the generated weights do not follow a uniform distribution in $\mathcal{W}_m$. The resulting distribution from this random sampling is shown in Figure 9.5b. It is skewed towards one vertex of the space (the dimension sampled). Generating uniformly all the w_i and then normalising does not either generate an uniform distribution (the solutions are skewed away from the vertices of the space).

Algorithm 9.2 [155] shows an algorithm to sample uniformly in the simplex $\mathcal{W}_m$. It first generates w_i for $0 \leq i \leq m$ according to the distribution

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 - e^{-x} & \text{if } x \geq 0 \end{cases}$$

To do this, we use the fact that if X is uniformly distributed, then the random variable $F^{-1}(X)$ is distributed according to F. The computation line 3 generates the w_i using

```
1  Function DasAndDennis(m, β)
       Data: Two positive integers m and β.
       Result: A set of points in the simplex W_m.
2      S ← ∅
3      T ← array of size m + 1
4      DasAndDennisRec(m, β, S, 0)
5      return S
6  Function DasAndDennisRec(m, β, S, i, T)
       Data: Three integers m,β and i, a set S.
       Result: No output, but points are added in the set S.
7      if i = m then
8          T[m] ← 1 − Σ_{j=0}^{m−1} T[j]
9          S.add(Copy(T))
10     else
11         k ← 0
12         while k/β + Σ_{j=0}^{i−1} T[j] ≤ 1 do
13             T[i] ← k/β
14             DasAndDennisRec(m, β, S, i + 1, T)
15             k ← k + 1
```

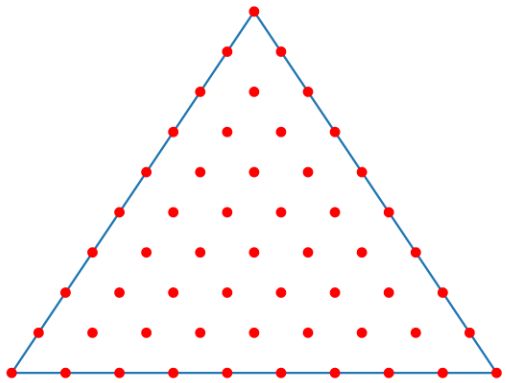
Algorithm 9.3: Generation of points in a grid in the simplex [148]

this property. The weights are then divided by their sum, to ensure that the sum of the returned vector is equal to 1. The proof of the correctness of this algorithm can be found in [155].

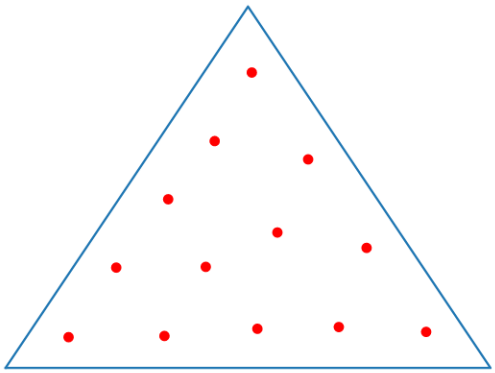
Remark. *There is another simple way to uniformly generate weights presented in [149]. The idea is to take the interval* $[0, 1]$*, cut it at random places, and return the length of the cut intervals (which necessarily sum to 1). First values* $w_1, \ldots, w_m$ *are generated uniformly in* $[0, 1]$*. Then the values* $w'_0, \ldots, w'_{m+1}$ *are computed by sorting the values* $0, w_1, \ldots, w_m, 1$*. The vector* $(w'_1 - w'_0, \ldots, w'_{m+1} - w'_m)$ *is uniformly distributed in* $\mathcal{W}_m$*. The complexity is* $\mathcal{O}(m \log m)$*, compared to the* $\mathcal{O}(m)$ *complexity of Algorithm 9.2.*

9.4.2 Das and Dennis' Generation

In [148], the authors proposed a weight generation algorithm that generates evenly spaced points (in a grid) of the simplex $\mathcal{W}_m$. It is presented in Algorithm 9.3. The algorithm takes as input the number of dimensions m (to be sampled in the simplex $\mathcal{W}_m$), and an integer β used to define the spacing of the points. It generates points in the simplex



(a) Das and Dennis' generation



(b) Adapted Lloyd's algorithm generation

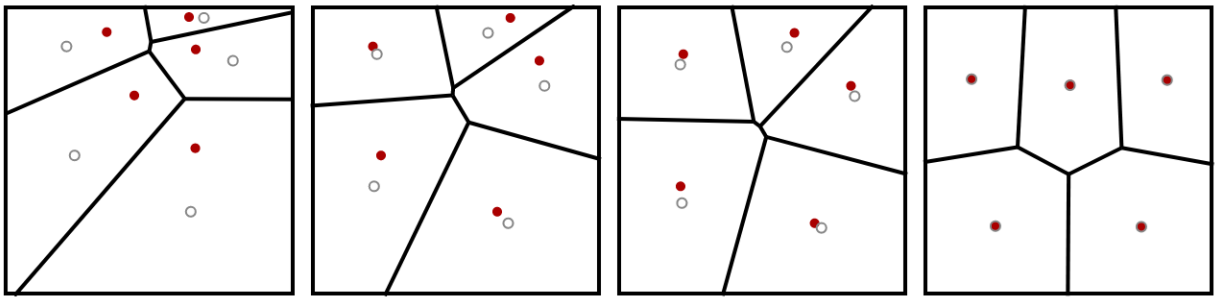
Figure 9.6 – Examples of diverse weights generation.

in a structured way, as shown in Figure 9.6a. The parameter β defines how spaced the points are. For example, in Figure 9.6a, $\beta = 9$, so that there are 10 points $(\beta + 1)$ on the sides of the triangle. Algorithm 9.3 recursively constructs the points in the array T, and when m dimensions are set (condition in line 7), the last dimension is set so that the sum of the values in T is equal to 1. The loop in line 12 enumerates the possible values for the i-th dimension and performs the recursive calls (to fill the remaining dimensions).

This procedure generates evenly spaced points. However, it is not possible to choose exactly how many points are generated. On $\mathcal{W}_m$, $\binom{m+\beta}{\beta}$ points are generated. Also, as the dimension increases, there are more and more points in the borders of the simplex (in the sides of the triangle for $\mathcal{W}_2$, or in the surface of the tetrahedron for $\mathcal{W}_3$). This flaw has been tackled in [150] by applying a *layer-wise* procedure.

9.4.3 Adapted Lloyd's Algorithm

The previous approach generates diverse weights, but the number of weights cannot be chosen. We propose here an approach to generate a selected number of weights that covers the space. This approach is an adaptation of Lloyd's Algorithm [177]. Lloyd's algorithm, also called Voronoi iteration, is an iterative algorithm that generates evenly spaced points. Initially, it randomly generates k points in the given space, computes the Voronoi diagram, and updates the k points to be in the centre of the Voronoi cells. This updating of the points (computing the centre of the Voronoi cell) is performed until the procedure converges, and then the points are returned. An example of execution in the unit square is shown in Figure 9.7.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 15

Figure 9.7 – Iterations of Lloyd's algorithm. In the last image, the centres of the Voronoi cells are close to the points: the algorithm converged. [3]

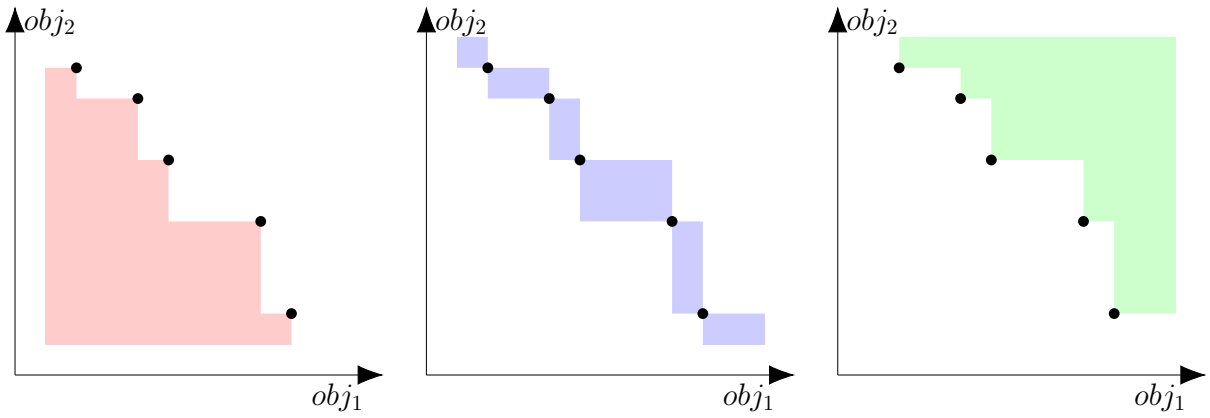
1 **Function** AdaptedLloyd(m, k, M)
  Data: Three integers m, k and M.
  Result: A set of k points in the simplex $\mathcal{W}_m$.
2 $\beta \leftarrow$ smallest integer such that $\binom{m+\beta}{\beta} \geq M$
3 $\mathcal{S} \leftarrow$ DansAndDennis(m, β)
4 DasAndDennisRec$(m, \beta, \mathcal{S}, 0)$
5 **return** K-Means$(\mathcal{S}, k)$

Algorithm 9.4: Generation of points in a grid in the simplex.

As the dimension increases, the computation of the Voronoi diagram becomes increasingly complex [162]. We remark that Lloyd's algorithm is a continuous version of the k-means algorithm. To generate weights, we propose Algorithm 9.4, an adaptation of Lloyd's algorithm by discretisation. A number of points M to be generated during the discretisation has to be chosen by the user. Then, at least M weights are generated using Das and Dennis' weight generation (the bound β is chosen accordingly in line 2). This discretises the simplex space. The choice of the bound M depends on the desired running time. A large value of M means that more time is spent in the next step, but with a better approximation. As the weights only need to be generated once, this running time is not an issue. Using this discretisation, the algorithm performs a k-means clustering, returning k points: our k diverse weights of the simplex $\mathcal{W}_m$.

As an example, Figure 9.6b shows the 13 points generated by Algorithm 9.4. The weights are not on the boundary of the simplex and are well distributed in the inside.

3. By Dominik Moritz - Own work, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=26443219`



(a) Dominated sub-space (b) Diversification sub-space (c) Intensification sub-space

Figure 9.4' – Graphical representation of the dominated, and non-dominated solutions in a bi-objective maximisation problem as in Figure 9.4. The dots are the points in $\mathcal{S} = \{(1,9),(3,8),(4,6),(7,4),(8,1)\}$.

9.5 Pareto Front Optimisation

The approach we propose in this chapter to find diverse solutions in multi-objective problems is a two-step procedure based on the PostHoc approach [28]. First, a set of solutions is generated (in our case, a non-dominated set of solutions, ideally the Pareto set). This set does not have to be diverse, but it should cover a large part of the space. Second, the k desired solutions are extracted from this set. This step should find diverse solutions.

In this section, we study the first part of this two-step approach. This step is where the problem is solved. There are two goals: to find *good* solutions, and to find multiple solutions *covering* the solution space. These two goals are similar to the exploration/ exploitation in several learning algorithms (such as Monte-Carlo Tree Search or Markov Decision Process). We want to improve good solutions previously found, but we also want to explore unseen parts of the space that may contain other good solutions.

9.5.1 Multi-Objective LNS

One of the first approaches to find a good non-dominated set of solutions in CP was presented in [156]. It adapts a Large Neighborhood Search (LNS) to multi-objective optimisation. LNS is a meta-heuristic improving solutions by destructing some parts of a previously found solution and reconstructing it the best possible way.

```
1 Function MO-LNS(P, limit, sublimit)
      Data: A multi-objective COP P = ⟨X, D, C, {obj_1, ..., obj_m}⟩, limit and
            sublimit two limits (such as the running time or number of
            backtracks).
      Result: A set of non-dominated solutions of P
2     S ← {FindOneSolution(P)}
3     P ← P ∧ pareto(obj_1, ..., obj_m, S)
4     while limit is not met do
5         s ← ChooseSolution(S)
6         S' ← ImproveLNS(P, s, sublimit)
7         S ← NonDominated(S ∪ S')
8         P ← P ∧ pareto(obj_1, ..., obj_m, S)
9     return S
```

Algorithm 9.5: Outline of the MO-LNS procedure.

The authors remark that the search space is divided into three sub-spaces, shown in Figure 9.4': the *dominated* sub-space (in Figure 9.4'a) contains all the solutions worse than those previously found in $\mathcal{S}$. They are removed using the Pareto constraint. The *diversification* sub-space (in Figure 9.4'b) contains all the solutions that are not comparable to those in $\mathcal{S}$. It is important to find these solutions to cover more of the search space. Finally, the *intensification* sub-space (in Figure 9.4'c) contains all the solutions that dominate at least one solution of $\mathcal{S}$. This sub-space should be explored to improve the current solutions.

Multi-Objective LNS (MO-LNS) presented in [156] iteratively improves a set $\mathcal{S}$ by selecting a solution, and using it in the LNS procedure to find a better solution. Algorithm 9.5 gives the outline of MO-LNS. Two limits must be chosen: *limit* to stop the computations, and *sublimit* to stop the iterations of the LNS. The algorithm starts with an initial solution, and adds the `pareto` constraint to the problem. Then, iterations are performed to improve the set $\mathcal{S}$. First, a solution s is selected using ChooseSolution. Then, this solution is improved (or diversified) using the LNS algorithm (adapted to the CP framework). At the end of the iteration, the non-dominated set of solutions $\mathcal{S}$ is updated with all the new solutions. This algorithm uses two important functions: ChooseSolution and ImproveLNS.

ChooseSolution

Due to the local search performed by the LNS, ImproveLNS finds solutions close to the one selected by ChooseSolution. A naive approach to select the solution would be to pick one at random from $\mathcal{S}$. However, as the authors remarked, this has a major flaw. Due to the locality effect of ImproveLNS, the set $\mathcal{S}$ may not be uniformly distributed in the space. Then, if the solution is chosen at random, it will with high probability be in sub-spaces of $\mathcal{S}$ with high density of solutions already found. Then new nearby solutions will be generated, increasing the density of $\mathcal{S}$ in that sub-space.

To avoid over-sampling some sub-spaces, a weight can be randomly (and uniformly) generated from $\mathcal{W}_{m-1}$ and used to pick the solution from $\mathcal{S}$. To pick a solution from a weight u, it is possible to pick the solution that is closest to the line defined by the vector u. This ensures that all sub-spaces are given the same chance.

ImproveLNS

When a solution s is chosen, an LNS iteration is performed to improve it. However, as we saw in Figures 9.4'b and 9.4'c, an improvement of s could mean either intensification (i.e. a dominant solution), or diversification (i.e. non-comparable solutions). To find solutions in the intensification sub-space, a constraint is added to the problem stating that the solution found should dominate s. To find solutions in the diversification space, no constraint is added. To alternate between diversification and intensification (as in exploration/exploitation), an LNS iteration will search in the diversification sub-space half of the time, and in the intensification sub-space the rest of the time. This has been shown experimentally to greatly improve the set $\mathcal{S}$.

9.5.2 Weighted Sum Strategy

The approach presented in this section is a personal communication [158] that does not appear in any publication. I present it here with the author's permission.

A simple way to deal with multi-objective problems is to transform them into single-objective optimisation (using a weighted sum for example), and solve them using the usual approach. However, this may not be possible when the objectives cannot be weighted. In [158], the author proposes a search strategy that uses this transformation (a weighted sum of the objectives) to improve the optimisation of the non-dominated set of solutions.

1 **Function** *WeightedSumStrategy*$(\mathcal{X}, \omega, score_1, \ldots, score_m)$
  Data: A set of variables $\mathcal{X}$, ω a point in the simplex $\mathcal{W}_{m-1}$, and m score functions $score_i : \mathcal{X} \times \mathbb{N} \to \mathbb{R}$.
  Result: A decision to perform during the search.
2 $(X, v) \leftarrow \underset{X \in \mathcal{X}, v \in \mathcal{D}(X)}{\operatorname{argmax}} \left(\sum_{i=1}^{m} \omega_i \cdot score_i(X, v) \right)$
3 **return** Decision$(X = v)$

Algorithm 9.6: Strategy weighting variable/value pairs depending on scores.

1 **Function** *ScoreSolving*$(\mathcal{P}, limit, sublimit, \mathcal{W}, score_1, \ldots, score_m)$
  Data: A multi-objective COP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \{obj_1, \ldots, obj_m\} \rangle$, *limit* and *sublimit* two limits (such as the running time or number of backtracks), a set of weights $\mathcal{W} \subset \mathcal{W}_{m-1}$, and m score functions $score_i : \mathcal{X} \times \mathbb{N} \to \mathbb{R}$.
  Result: A set of non-dominated solutions of $\mathcal{P}$
2 $\mathcal{S} \leftarrow \emptyset$
3 **while** *limit is not met* **do**
4 $\omega \leftarrow$ NextWeight$(\mathcal{W})$
5 $strat \leftarrow$ WeightedSumStrategy$(\mathcal{X}, \omega, score_1, \ldots, score_m)$
6 $\mathcal{S}' \leftarrow$ FindSolutions $(\mathcal{P}, strat, sublimit)$
7 $\mathcal{S} \leftarrow$ NonDominated$(\mathcal{S} \cup \mathcal{S}')$
8 $\mathcal{P} \leftarrow \mathcal{P} \wedge \texttt{pareto}(obj_1, \ldots, obj_m, \mathcal{S})$
9 **return** $\mathcal{S}$

Algorithm 9.7: Iteratively solving the problem using a weighted strategy to find good multi-objective solutions.

This approach assumes that there is a way (or an approximation) to score the variable/value pairs to optimise a given objective. It assumes that for the problem, there is a function $score_1$ such that the value v for variable X that maximises $score_1(X, v)$ is the best decision to optimise objective obj_1. Using these scores, the author designs a search strategy performing a weighted sum of the scores. Algorithm 9.6 presents the search strategy. The decision performed by the strategy is the couple variable/value that maximises the weighted sum of the scores on each dimension of the problem. For example, in a two dimension problem, with a weight $\omega = (1/2, 1/2)$, the strategy WeightedSumStrategy chooses the variable/value pair that maximises the sum of the scores on the two dimensions.

Algorithm 9.7 presents the main solving algorithm using WeightedSumStrategy. As with MO-LNS, it is an iterative process. A *limit* is set on the number of iterations

(or time). A weight is chosen from a set $\mathcal{W}$ of weights (in the $m-1$-simplex). This weight is used in the weighted strategy. This strategy is then used to find solutions to the multi-objective problem (with the `pareto` constraint) until a *sublimit* is reached (number of solutions found, number of backtracks performed, or time limit). These newly found solutions are added to the non-dominated set of solutions $\mathcal{S}$.

The set of weights $\mathcal{W}$ must be chosen to be diverse. In this case, Das and Dennis' generation is very good because it covers well the whole space (including the boundaries). There is no issue in not having a chosen number of weights, as there will be many iterations of the algorithm. In two dimensions, the weights are of the form $(t, 1-t)$ for $t \in [0, 1]$. In this case, a small optimisation can be done when selecting the next weight. At the beginning, t is increased from 0 to 1, but at $t = 1$, instead of cycling back from 0, it is possible to go back from $t = 1$ to $t = 0$. This way, as we previously found good solutions using the weight $(0, 1)$, the `pareto` constraint will prune a larger part of the search space, thus finding even better solutions.

This solving algorithm using WeightedSumStrategy is efficient when good scores are available for the problem. For example, in a knapsack problem, items have a cost (to be maximised) and a weight (the total weight of the selected items is bounded). A good score to maximise the cost of the selected items is the efficiency: the cost divided by the weight. If more dimensions are added to the problem, this efficiency can be defined for each of them: dividing the value of the dimension by the weight. However, the existence of weights is a strong assumption. Sometimes the variables are not easy to rank. Also, a weighted sum heavily depends on the range of the scores. If in one dimension, $score_1$ gives scores ranging from 0 to 10, and in another dimension, $score_2$ gives scores ranging from 0 to 10000, the second dimension will have a greater impact on WeightedSumStrategy.

9.5.3 Wavering Strategy

The strategy presented in the previous section uses scores to select the pair variable/ value on which to branch, however such scores may not always be available or comparable. To get around this issue, we propose a new meta-search strategy. Our search strategy, Wavering, only assumes that there exist search strategies that optimise each dimension, without assuming that these search strategies give a score to the variables. We use these sub-strategies as a black-box. We use weights, seen as a probability distribution, to randomly choose the strategy used. The strategy is called Wavering because it wavers between several strategies.

1 **Function** *Wavering*$(\mathcal{X}, \omega, strat_1, \ldots, strat_m)$
  Data: A set of variables $\mathcal{X}$, ω a point in the simplex $\mathcal{W}_{m-1}$, and m search strategies $strat_i$.
  Result: A decision to perform during the search.
2 $strat \leftarrow$ Random $([strat_1, \ldots, strat_m], \omega)$
3 **return** $strat(\mathcal{X})$

Algorithm 9.8: Strategy picking at random a sub-strategy optimising a given dimension.

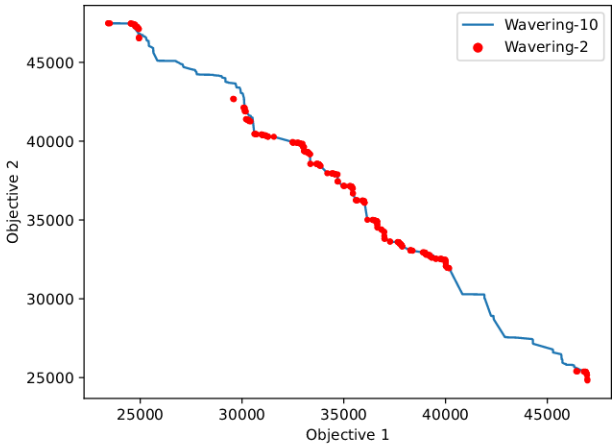


Figure 9.8 – Non-dominated sets of solutions in a random knapsack instance with two objectives, with different numbers of weights (generated with $\beta = 2$ or 10). Each point is a solution in the objective space. Wavering-2 uses the weights $(0, 1), (1/2, 1/2)$ and $(1, 0)$ and Wavering-10 uses the 11 weights $(t, 1-t)$ for $t \in \{0, 1/10, \ldots, 9/10, 1\}$.

We present the Wavering search strategy in Algorithm 9.8. It simply uses the weight ω as a probability distribution to randomly selects the strategy *strat* used to make the decision. For example, in two dimensions, if the weight is $\omega = (0.9, 0.1)$, then the first strategy will be selected 90% of the time, and the second strategy will be picked 10% of the time.

The global solving algorithm is similar to Algorithm 9.7. The only difference is the search strategy in line 5 and the parameters (the strategies for each dimension instead of the scores). The weights can be chosen using Das and Dennis' approach. There should be enough weights on the inside of the simplex.

This strategy uses the same idea as WeightedSumStrategy, but does not assume that scores exist. Instead, it uses randomness to choose between the two strategies. It is also non-deterministic, so after a restart, the strategy may not go into the same space. This means, for example, that with the weight $(1/2, 1/2)$ in two dimensions, the strategy

does not only find a solution that optimises the sum of the two objectives, but also nearby solutions. This is illustrated in Figure 9.8. The points in orange are the set of non-dominated solutions returned when using WAVERING with only the three weights $(0,1),(1/2,1/2)$ and $(1,0)$ (using $\beta = 2$)). We can see that there are solutions that are very good for obj_1 (because of the weight $(1,0)$), some solutions are very good for obj_2 (because of the weight $(0,1)$), and several solutions are a compromise between the two objectives (because of the weight $(1/2,1/2)$). We can remark that there is no randomness when using the weights $(1,0)$ and $(0,1)$ (only the strategy that optimises one objective is used), so the solutions are thus more often located on the edge of the front. With the randomness, the space covered by the search strategy (with multiple restarts) is very large. The line in blue is the WAVERING approach with more weights (11 weights with $\beta = 10$ in Das and Dennis' generation). Using more weights fills in the gaps in the front that were missing when only three weights were used. The whole front is well sampled with our approach.

9.5.4 Conclusion on Pareto Optimisation

We have presented three approaches to find good non-dominated sets of solutions. It is interesting to note that these three approaches use weights in the simplex for different purposes. However, they are always used to guide the search to a new sub-space. In MO-LNS, these weights are used to find solutions that cover the space, to improve every part of the space. In WEIGHTEDSUMSTRATEGY they are used in a weighted sum to aggregate scores. In WAVERING they are used as a probability distribution to randomly pick a sub-search strategy.

9.6 Solution Set Extraction

The previous section presented how to find a good non-dominated set of solutions in the whole search space. The second step of the POSTHOC approach we use here is to extract few diverse solutions from this set. In this section we first present exact algorithms in two dimensions . We then propose a simple approach based on weights to extract solutions in higher dimension.

9.6.1 Efficient Algorithms for Two Objectives

When restricted to two objectives, some multi-objective problems become easier. This is the case for the subset selection problem.

Hypervolume Indicator

To optimise the hypervolume indicator $\mathcal{H}yp$, algorithms in time $\mathcal{O}(n(k + \log n))$ were proposed in [147, 153]. They find the subset of size k that maximises the hypervolume indicator from a non-dominated set of solutions of size n. This algorithm is an improvement on the dynamic programming approach presented in [145]. We present an adaptation of the dynamic programming approach because it is simpler and can easily be extended to other metrics (such as the minimum distance or the representativeness). This dynamic approach uses the efficient computation of the hypervolume for a bi-objective problem [146].

Property 7 ([146])**.** *Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a set of solutions sorted by increasing first dimension, and r be a reference point. Then*

$$\mathcal{H}yp(\mathcal{S}) = (s_1[0] - r[0])(s_1[1] - r[1]) + \sum_{i=2}^{n}(s_i[0] - s_{i+1}[0])(s_i[1] - r[1]) .$$

The dynamic approach defines a recursive function $f(i, l)$ equal to the maximum hypervolume contribution possible with l solutions without considering the area before $s_i[0]$ (where r is s_0 by convention). This function can be computed recursively by the following formula:

$$f(i, l) = \max_{i<j\leq n}(s_j[0] - s_i[0])(s_j[1] - r[1]) + f(j, l - 1)$$

This recursive formula searches for the next solution s_j to take, and recursively calls $f(j, l-1)$ (i.e. the maximum hypervolume of $l-1$ solutions without considering the area before $s_j[0]$). Then $f(0, k)$ gives the maximum hypervolume of a set of k solutions. As usual for dynamic programming approaches, a subset that achieves the best value can be easily retrieved.

This dynamic approach can be extended to metrics other than the hypervolume. For example if the minimum distance δ between the solutions is considered, the formula for f becomes:

$$f(i, l) = \max_{i<j\leq n} \min\left(\delta(s_i, s_j), f(j, l - 1)\right) .$$

This approach has a complexity of $\mathcal{O}(n^2 k)$. When the set of solutions $\mathcal{S}$ is not too

large (less than tens of thousands of solutions), the best subset can be computed quickly.

Representative Set

Representative solutions were introduced in [157]. In [151] the authors propose an algorithm to find the most representative subset of solutions when the Pareto front is known. We recall that the radius of a set of solutions $\mathcal{S}$ from a Pareto front (or an approximation of it) $\mathcal{F}$ is

$$\max_{p' \in \mathcal{F}} \min_{p \in \mathcal{S}} \delta(p, p') \,.$$

This radius measures the maximum distance between a point in the front and its representative (the closest solution) in the selected subset.

In [151], the algorithm for finding the most representative subset is divided into two algorithms. First, given a maximum radius, an algorithm generates a subset satisfying that radius (or nothing if no such subset exists). Then a dichotomic search on the value of the radius allows to find the optimal one.

9.6.2 Higher Dimension Algorithm

In higher dimension (more than 2) the problem of finding the most diverse subset is more difficult. In [157] the authors show that if the Pareto front is known, the problem is NP-complete, but when it is unknown it is Σ_2^P-complete (i.e. one class of complexity harder).

In our case, this is the second step of the procedure (first generating good solutions, and then extracting diverse solutions). We propose to use an approach similar to the one used in MO-LNS's solution selection. If k solutions are desired by the user, we generate k weights, for example using the adapted Lloyd's approach. We then use these weights to select solutions. A weight is a vector in the objective space. This vector supports a line. We select the solution that is the closest to the line defined by the vector. Let s be a solution (a vector) in the objective space, and ω be a weight. We note u the normalised (in euclidean distance) vector of ω, i.e. $\omega / \|\omega\|$. The distance between s and the line defined by ω is $dist(s, \omega) = \|s - (s \cdot u)u\|$. where $\cdot$ is the vector dot product and $\|v\|$ is the norm of the vector v.

For each of the k chosen weights, we select the solution s that minimises $dist(s, \omega)$. It is possible that some solutions are the same (if some part of the space is very sparse), but

in this case, it is possible to randomly generate new weights until the desired number of solutions is reached.

9.7 Conclusion

In this chapter, we experimented our approach on a real-life problem, which made us adapt our ideas to multi-objective problems. We presented an application to planning problems aimed at helping a non-expert user (for example, a prefect) to make decisions. In this setting, multiple objectives can be defined, and there exist no *optimal* solution on all of the objectives. Also, the users must choose between solutions because it has a heavy impact on people. Multiple good solutions should be presented to them.

The approach we propose to find diverse solutions is two step and inspired by the PostHoc approach. First, a good non-dominated set of solutions is found using the solver. In optimisation problems, the search strategy is very important, so we propose a new search strategy Wavering that uses sub-search strategies that optimise each dimension separately. This strategy covers the objective space well, and generates good solutions.

Then, in a second step, we propose to generate weights using an adaptation of Lloyd's algorithm, and use these weights to extract solutions. This step does not use a CP solver, as the solutions are already found.

Our approach fits well within the CP framework. In fact, we defined a meta-search strategy for multi-objective optimisation. This is similar to the search strategies we presented in Part III, but adapted to multi-objective optimisation. The diversity provided by using a probabilistic approach allows different parts of the search space to be searched, while still orienting the search towards good solutions. It is also possible to customise it, by choosing the weights. It can be used to model the users' preferences to search in certain spaces. In the same way, the second step of the approach, the solution set extraction, can be easily tuned to return more solutions in a given space.

Chapter 10

Conclusion

In this thesis, we have shown how to add diversity to the solutions returned by a CP solver. To do this, we used probabilistic approaches. Adding randomness to the search breaks the rigid backtrack search solving algorithm, and allows the solver to explore different spaces.

In constrained problems, a first way to add randomness is to randomly sample solutions. With samplers, everything depends on the guarantees of the distribution of the solutions. The most commonly desired guarantee is uniformity of the distribution, but the distribution can also be specified by a user. Designing a sampler with guarantees (and proving these guarantees) is a difficult task. Some other samplers are not guaranteed not to be uniform, but they can return many more solutions in the same time. In CP, the design of a sampler is made even more difficult by the variety of constraints (compared to the clauses in SAT).

The sampler we proposed, TableSampling, is designed to provide randomness in a reasonable time. The distribution is not uniform, because the focus is on the running time. We obtain a good trade-off between randomness quality and computation time. Instead of a perfectly uniform but costly sampler, a good approximation with a faster running time may be more useful for the user. In a *FairAI* approach, the randomness in the decision also ensure fairness between the solutions. Today, decision performed by algorithms impact people, and in this case, randomness ensures that there is no bias of the solver towards certain solutions. This way, the same people will not always be aggrieved.

We adopted the same practical approach for pattern mining and feature model configurations. We used search strategies to find diverse solutions quickly. These solutions are not uniformly distributed, but by adjusting the randomness, they can be very diverse. We even showed that a uniform distribution does not always provide a good diversity of solutions, due to the way solutions are distributed in the search space. We encourage users and modellers to use search strategies designed for their problem (with a specific diversity measure) to find diverse solutions. If this is not satisfactory, more powerful approaches

(such as diversity constraints, or weighted sampling) can be used.

In this thesis, we have also shown that randomness is a very powerful tool. This may deter users at first but we have shown that on average there are great guarantees (approximation factors, and average lower bounds). Moreover, the lack of strong guarantees can be overcome by applying a post-processing step to the randomly generated solutions. This produces very diverse sets of solutions much faster than exhaustive search, which is generally not even applicable.

We have applied diversity to a real problem. This is an ongoing collaboration, and the goal is to present solutions to a real user. This user is the only one who can tell if the solutions are diverse or not. The evaluation metrics are only here as a modelling tool. It would be very interesting to see how a user reacts to the solutions and what their requirements are. Further research should be conducted on designing meaningful visualisation tools to show as much information as possible, as quickly as possible.

Diversity should be applied whenever real-life impactful decisions are made, in particular when solvers are used in cases where the solutions have consequences on people, such as in disaster management. This means that the approaches should be extended to as many applications as possible. In this thesis, we have always taken generic approaches (using the model as a black-box), but we have also used domain knowledge to improve the search of the solutions (such as commonalities in feature models). It would be interesting to develop a framework that allows the users to specify their problem, but also the desired properties of the solutions (diverse, covering, optimal), without taking into account the implementation (diversity constraints, random search, Pareto optimisation). This is more than just an API on top of a CP solver, it would require to being able to model domain knowledge in a generic way, and to add it to the CP solving process.

A first step towards this framework would be to consider a specific domain and see how domain knowledge can be represented. In the introduction of this thesis, I presented video game randomisers. This is an example of a domain that requires randomisation (changing which items are where) under the constraints that the game should still be able to be completed. Also, domain knowledge should force the randomisation to be *interesting*, so that all the powerful items are not at the beginning of the game. There are many games that already have randomisers, but they are usually developed by players in their own community, without using randomisers from other games as a starting point. There is no common framework to facilitate the development process. The use of constrained randomisation and diversity can benefit to all the players.

BIBLIOGRAPHY

Author's Publications

[1] Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme, « Solution Sampling with Random Table Constraints », *in*: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, ed. by Laurent D. Michel, vol. 210, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 56:1–56:17, URL: https://doi.org/10.4230/LIPIcs.CP.2021.56.

[2] Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme, « Solution sampling with random table constraints », *in*: *Constraints An Int. J.* 27.*4* (2022), pp. 381–413, URL: https://doi.org/10.1007/s10601-022-09329-w.

[3] Mathieu Vavrille et al., *Linear Time Computation of Variation Degree and Commonalities on Feature Diagrams*, tech. rep. RR-2023-01-DAPI, Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, F-44000 Nantes, France, Jan. 2023, URL: https://hal.science/hal-03970237.

Constraint Programming and SAT

[4] Özgür Akgün et al., « Instance Generation via Generator Instances », *in*: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, ed. by Thomas Schiex and Simon de Givry, vol. 11802, Lecture Notes in Computer Science, Springer, 2019, pp. 3–19, URL: https://doi.org/10.1007/978-3-030-30048-7_1.

[5] Carlos Ansótegui et al., « Community Structure in Industrial SAT Instances », *in*: *J. Artif. Intell. Res.* 66 (2019), pp. 443–472, URL: https://doi.org/10.1613/jair.1.11741.

[6] Pierre Berlandier, « Improving domain filtering using restricted path consistency », *in*: *Proceedings the 11th Conference on Artificial Intelligence for Applications*, IEEE, 1995, pp. 32–37, URL: https://doi.org/10.1109/CAIA.1995.378792.

[7] Christian Bessière, « Arc-Consistency and Arc-Consistency Again », *in*: *Artif. Intell.* 65.*1* (1994), pp. 179–190, URL: https://doi.org/10.1016/0004-3702(94)90041-8.

[8] Christian Bessière and Jean-Charles Régin, « Refining the Basic Constraint Propagation Algorithm », *in*: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, ed. by Bernhard Nebel, Morgan Kaufmann, 2001, pp. 309–315, URL: https://dl.acm.org/doi/10.5555/1642090.1642133.

[9] Armin Biere et al., eds., *Handbook of Satisfiability - Second Edition*, vol. 336, Frontiers in Artificial Intelligence and Applications, IOS Press, 2021, ISBN: 978-1-64368-160-3, URL: https://doi.org/10.3233/FAIA336.

[10] Jérémie Du Boisberranger et al., « When is it worthwhile to propagate a constraint? A probabilistic analysis of AllDifferent », *in*: *Proceedings of the 10th Meeting on Analytic Algorithmics and Combinatorics, ANALCO 2013, New Orleans, Louisiana, USA, January 6, 2013*, ed. by Markus E. Nebel and Wojciech Szpankowski, SIAM, 2013, pp. 80–90, URL: https://doi.org/10.1137/1.9781611973037.10.

[11] Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette, « XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems », *in*: *CoRR* abs/1611.03398 (2016), URL: http://arxiv.org/abs/1611.03398.

[12] Frédéric Boussemart et al., « Boosting Systematic Search by Weighting Constraints », *in*: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, ed. by Ramón López de Mántaras and Lorenza Saitta, IOS Press, 2004, pp. 146–150.

[13] Chiu Wo Choi et al., « Finite Domain Bounds Consistency Revisited », *in*: *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, ed. by Abdul Sattar and Byeong-Ho Kang, vol. 4304, Lecture Notes in Computer Science, Springer, 2006, pp. 49–58, URL: https://doi.org/10.1007/11941439_9.

[14] Martin Davis, George Logemann, and Donald W. Loveland, « A machine program for theorem-proving », *in*: *Commun. ACM* 5.*7* (1962), pp. 394–397, URL: https://doi.org/10.1145/368273.368557.

[15] Jordan Demeulenaere et al., « Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets », *in*: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, ed. by Michel Rueher, vol. 9892, Lecture Notes in Computer Science, Springer, 2016, pp. 207–223, URL: https://doi.org/10.1007/978-3-319-44953-1_14.

[16] Grégoire Dooms, Yves Deville, and Pierre Dupont, « CP(Graph): Introducing a Graph Computation Domain in Constraint Programming », *in*: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, ed. by Peter van Beek, vol. 3709, Lecture Notes in Computer Science, Springer, 2005, pp. 211–225, URL: https://doi.org/10.1007/11564751_18.

[17] Ian P. Gent et al., « Discriminating Instance Generation for Automated Constraint Model Selection », *in*: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, ed. by Barry O'Sullivan, vol. 8656, Lecture Notes in Computer Science, Springer, 2014, pp. 356–365, URL: https://doi.org/10.1007/978-3-319-10428-7_27.

[18] Daniel Godard, Philippe Laborie, and Wim Nuijten, « Randomized Large Neighborhood Search for Cumulative Scheduling », *in*: *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, ed. by Susanne Biundo, Karen L. Myers, and Kanna Rajan, AAAI, 2005, pp. 81–89, URL: https://cdn.aaai.org/ICAPS/2005/ICAPS05-009.pdf.

[19] Carla P. Gomes, Bart Selman, and Henry A. Kautz, « Boosting Combinatorial Search Through Randomization », *in*: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, ed. by Jack Mostow and Chuck Rich, AAAI Press / The MIT Press, 1998, pp. 431–437.

[20] Arnaud Gotlieb and Dusica Marijan, « Using Global Constraints to Automate Regression Testing », *in*: *AI Mag.* 38.*1* (2017), pp. 73–87, URL: https://doi.org/10.1609/aimag.v38i1.2714.

[21] Robert M. Haralick and Gordon L. Elliott, « Increasing Tree Search Efficiency for Constraint Satisfaction Problems », *in*: *Artif. Intell.* 14.*3* (1980), pp. 263–313, URL: https://doi.org/10.1016/0004-3702(80)90051-X.

[22] Warwick Harvey and Joachim Schimpf, « Bounds consistency techniques for long linear constraints », *in*: *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a workshop of CP*, 2002, pp. 39–46.

[23] Emmanuel Hebrard, « Robust solutions for constraint satisfaction and optimisation under uncertainty », PhD thesis, University of New South Wales, Sydney, Australia, 2007, URL: http://handle.unsw.edu.au/1959.4/40765.

[24] Emmanuel Hebrard et al., « Finding Diverse and Similar Solutions in Constraint Programming », *in*: *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, ed. by Manuela M. Veloso and Subbarao Kambhampati, AAAI Press / The MIT Press, 2005, pp. 372–377, URL: https://cdn.aaai.org/AAAI/2005/AAAI05-059.pdf.

[25] Emmanuel Hebrard et al., « Soft Constraints of Difference and Equality », *in*: *J. Artif. Intell. Res.* 41 (2011), pp. 97–130, URL: https://doi.org/10.1613/jair.3197.

[26] Marijn Heule et al., « Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads », *in*: *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, ed. by Kerstin Eder, João Lourenço, and Onn Shehory, vol. 7261, Lecture Notes in Computer Science, Springer, 2011, pp. 50–65, URL: https://doi.org/10.1007/978-3-642-34188-5_8.

[27] Willem Jan van Hoeve, « A Hyper-arc Consistency Algorithm for the Soft Alldifferent Constraint », *in*: *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ed. by Mark Wallace, vol. 3258, Lecture Notes in Computer Science, Springer, 2004, pp. 679–689, URL: https://doi.org/10.1007/978-3-540-30201-8_49.

[28] Linnea Ingmar et al., « Modelling Diversity of Solutions », *in*: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, AAAI Press, 2020, pp. 1528–1535, URL: https://ojs.aaai.org/index.php/AAAI/article/view/5512.

[29] Paul B. Jackson and Daniel Sheridan, « Clause Form Conversions for Boolean Circuits », *in*: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, ed. by Holger H. Hoos and David G. Mitchell, vol. 3542, Lecture Notes in Computer Science, Springer, 2004, pp. 183–198, URL: https://doi.org/10.1007/11527695_15.

[30] Narendra Jussien, Olivier Lhomme, and SA Ilog, « Unifying search algorithms for CSP », *in*: (2002), Technical report 02-3-INFO, EMN.

[31] Michel Leconte, « A bounds-based reduction scheme for difference constraints », *in*: *Proceedings of the FLAIRS*, vol. 96, 1996.

[32] Christophe Lecoutre et al., « Last Conflict Based Reasoning », *in*: *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, ed. by Gerhard Brewka et al., vol. 141, Frontiers in Artificial Intelligence and Applications, IOS Press, 2006, pp. 133–137, URL: http://dx.doi.org/10.1016/j.artint.2009.09.002.

[33] Hongbo Li, Minghao Yin, and Zhanshan Li, « Failure Based Variable Ordering Heuristics for Solving CSPs (Short Paper) », *in*: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, ed. by Laurent D. Michel, vol. 210, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9:1–9:10, URL: https://doi.org/10.4230/LIPIcs.CP.2021.9.

[34] Alan K. Mackworth, « Consistency in Networks of Relations », *in*: *Artif. Intell.* 8.*1* (1977), pp. 99–118, URL: https://doi.org/10.1016/0004-3702(77)90007-8.

[35] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre, « The Smart Table Constraint », *in*: *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, ed. by Laurent Michel, vol. 9075, Lecture Notes in Computer Science, Springer, 2015, pp. 271–287, URL: https://doi.org/10.1007/978-3-319-18008-3_19.

[36] Laurent Michel and Pascal Van Hentenryck, « Activity-Based Search for Black-Box Constraint Programming Solvers », *in*: *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, ed. by Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, vol. 7298, Lecture Notes in Computer Science, Springer, 2012, pp. 228–243, URL: https://doi.org/10.1007/978-3-642-29828-8_15.

[37] Roger Mohr and Thomas C. Henderson, « Arc and Path Consistency Revisited », *in*: *Artif. Intell.* 28.*2* (1986), pp. 225–233, URL: https://doi.org/10.1016/0004-3702(86)90083-4.

[38] Matthew W. Moskewicz et al., « Chaff: Engineering an Efficient SAT Solver », *in*: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, ACM, 2001, pp. 530–535, URL: https://doi.org/10.1145/378239.379017.

[39] Nicholas Nethercote et al., « MiniZinc: Towards a Standard CP Modelling Language », *in*: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, ed. by Christian Bessiere, vol. 4741, Lecture Notes in Computer Science, Springer, 2007, pp. 529–543, URL: https://doi.org/10.1007/978-3-540-74970-7_38.

[40] Guillaume Perez et al., « Distribution Optimization in Constraint Programming », *in*: *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, Toronto, Canada, August 27-31, 2023*, ed. by Roland H. C. Yap, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 29:1–29:19, URL: https://doi.org/10.4230/LIPIcs.CP.2023.29.

[41] Gilles Pesant, Kuldeep S. Meel, and Mahshid Mohammadalitajrishi, « On the Usefulness of Linear Modular Arithmetic in Constraint Programming », *in*: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5-8, 2021, Proceedings*, ed. by Peter J. Stuckey, vol. 12735, Lecture Notes in Computer Science, Springer, 2021, pp. 248–265, URL: https://doi.org/10.1007/978-3-030-78230-6_16.

[42] Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini, « Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems », *in*: *J. Artif. Intell. Res.* 43 (2012), pp. 173–210, URL: https://doi.org/10.1613/jair.3463.

[43] Thierry Petit, Jean-Charles Régin, and Nicolas Beldiceanu, « A Θ(n) Bound-Consistency Algorithm for the Increasing Sum Constraint », *in*: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, ed. by Jimmy Ho-Man Lee, vol. 6876, Lecture Notes in Computer Science, Springer, 2011, pp. 721–728, URL: https://doi.org/10.1007/978-3-642-23786-7_54.

[44] Thierry Petit and Andrew C. Trapp, « Finding Diverse Solutions of High Quality to Constraint Optimization Problems », *in*: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, ed. by Qiang Yang and Michael J. Wooldridge, AAAI Press, 2015, pp. 260–267, URL: http://ijcai.org/Abstract/15/043.

[45] Steven D. Prestwich, Roberto Rossi, and S. Armagan Tarim, « Randomness as a Constraint », *in*: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, ed. by Gilles Pesant, vol. 9255, Lecture Notes in Computer Science, Springer, 2015, pp. 351–366, URL: https://doi.org/10.1007/978-3-319-23219-5_25.

[46] Charles Prud'homme and Jean-Guillaume Fages, « Choco-solver: A Java library for constraint programming », *in*: *Journal of Open Source Software* 7.*78* (2022), p. 4708, URL: https://doi.org/10.21105/joss.04708.

[47] Jean-Francois Puget, « A Fast Algorithm for the Bound Consistency of alldiff Constraints », *in*: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, ed. by Jack Mostow and Chuck Rich, AAAI Press / The MIT Press, 1998, pp. 359–366, URL: https://cdn.aaai.org/AAAI/1998/AAAI98-051.pdf.

[48] Philippe Refalo, « Impact-Based Search Strategies for Constraint Programming », *in*: *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ed. by Mark Wallace, vol. 3258, Lecture Notes in Computer Science, Springer, 2004, pp. 557–571, URL: https://doi.org/10.1007/978-3-540-30201-8_41.

[49] Jean-Charles Régin, « A Filtering Algorithm for Constraints of Difference in CSPs », *in*: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, ed. by Barbara Hayes-Roth and Richard E. Korf, AAAI Press / The MIT Press, 1994, pp. 362–367, URL: https://cdn.aaai.org/AAAI/1994/AAAI94-055.pdf.

[50] Francesca Rossi, Peter van Beek, and Toby Walsh, eds., *Handbook of Constraint Programming*, vol. 2, Foundations of Artificial Intelligence, Elsevier, 2006, ISBN: 978-0-444-52726-4, URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

[51] Francesca Rossi, Kristen Brent Venable, and Toby Walsh, « Preferences in Constraint Satisfaction and Optimization », *in*: *AI Mag.* 29.*4* (2008), pp. 58–68, URL: https://doi.org/10.1609/aimag.v29i4.2202.

[52] Irena Rusu, « NP-hardness of sortedness constraints », *in*: *CoRR* abs/1506.02442 (2015), URL: http://arxiv.org/abs/1506.02442.

[53] Tian Sang, Paul Beame, and Henry A. Kautz, « Heuristics for Fast Exact Model Counting », *in*: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ed. by Fahiem Bacchus and Toby Walsh, vol. 3569, Lecture Notes in Computer Science, Springer, 2005, pp. 226–240, URL: https://doi.org/10.1007/11499107_17.

[54] Yevgeny Schreiber, « Value-Ordering Heuristics: Search Performance vs. Solution Diversity », *in*: *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, ed. by David Cohen, vol. 6308, Lecture Notes in Computer Science, Springer, 2010, pp. 429–444, URL: https://doi.org/10.1007/978-3-642-15396-9_35.

[55] Christian Schulte and Peter J. Stuckey, « Efficient constraint propagation engines », *in*: *ACM Trans. Program. Lang. Syst.* 31.*1* (2008), 2:1–2:43, URL: https://doi.org/10.1145/1452044.1452046.

[56] João P. Marques Silva, « The Impact of Branching Heuristics in Propositional Satisfiability Algorithms », *in*: *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, ed. by Pedro Barahona and José Júlio Alferes, vol. 1695, Lecture Notes in Computer Science, Springer, 1999, pp. 62–74, URL: https://doi.org/10.1007/3-540-48159-1_5.

[57] João P. Marques Silva and Karem A. Sakallah, « GRASP - a new search algorithm for satisfiability », *in*: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, ed. by Rob A. Rutenbar and Ralph H. J. M. Otten, IEEE Computer Society / ACM, 1996, pp. 220–227, URL: https://doi.org/10.1109/ICCAD.1996.569607.

[58] Mate Soos, Karsten Nohl, and Claude Castelluccia, « Extending SAT Solvers to Cryptographic Problems », *in*: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, ed. by Oliver Kullmann, vol. 5584, Lecture Notes in Computer Science, Springer, 2009, pp. 244–257, URL: https://doi.org/10.1007/978-3-642-02777-2_24.

[59] Armagan Tarim, Suresh Manandhar, and Toby Walsh, « Stochastic Constraint Programming: A Scenario-Based Approach », *in*: *Constraints An Int. J.* 11.*1* (2006), pp. 53–80, URL: https://doi.org/10.1007/s10601-006-6849-7.

[60] S. Armagan Tarim et al., « Finding reliable solutions: event-driven probabilistic constraint programming », *in*: *Ann. Oper. Res.* 171.*1* (2009), pp. 77–99, URL: https://doi.org/10.1007/s10479-008-0382-6.

[61] G. S. Tseitin, « On the Complexity of Derivation in Propositional Calculus », *in*: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, ed. by Jörg H. Siekmann and Graham Wrightson, Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483, ISBN: 978-3-642-81955-1, URL: https://doi.org/10.1007/978-3-642-81955-1_28.

[62] Toby Walsh, « Stochastic Constraint Programming », *in*: *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, ed. by Frank van Harmelen, IOS Press, 2002, pp. 111–115, URL: https://doi.org/10.48550/arXiv.0903.1152.

[63] Hugues Wattez et al., « Refining Constraint Weighting », *in*: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, IEEE, 2019, pp. 71–77, URL: https://doi.org/10.1109/ICTAI.2019.00019.

Samplers

[64] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos, « Fast Sampling of Perfectly Uniform Satisfying Assignments », *in*: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, ed. by Olaf Beyersdorff and Christoph M. Wintersteiger, vol. 10929, Lecture Notes in Computer Science, Springer, 2018, pp. 135–147, URL: https://doi.org/10.1007/978-3-319-94144-8_9.

[65] Mihir Bellare, Oded Goldreich, and Erez Petrank, « Uniform Generation of NP-witnesses using an NP-oracle », *in*: *Electron. Colloquium Comput. Complex.* (1998), URL: https://eccc.weizmann.ac.il/eccc-reports/1998/TR98-032/index.html.

[66] Sourav Chakraborty and Kuldeep S. Meel, « On Testing of Uniform Samplers », *in*: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, AAAI Press, 2019, pp. 7777–7784, URL: https://doi.org/10.1609/aaai.v33i01.33017777.

[67] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi, « A Scalable and Nearly Uniform Generator of SAT Witnesses », *in*: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ed. by Natasha Sharygina and Helmut Veith, vol. 8044, Lecture Notes in Computer Science, Springer, 2013, pp. 608–623, URL: https://doi.org/10.1007/978-3-642-39799-8_40.

[68] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi, « A Scalable Approximate Model Counter », *in*: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, ed. by Christian Schulte, vol. 8124, Lecture Notes in Computer Science, Springer, 2013, pp. 200–216, URL: https://doi.org/10.1007/978-3-642-40627-0_18.

[69] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi, « Balancing Scalability and Uniformity in SAT Witness Generator », *in*: *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, ACM, 2014, 60:1–60:6, URL: https://doi.org/10.1145/2593069.2593097.

[70] Supratik Chakraborty et al., « Distribution-Aware Sampling and Weighted Model Counting for SAT », *in*: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, ed. by Carla E. Brodley and Peter Stone, AAAI Press, 2014, pp. 1722–1730, URL: https://ojs.aaai.org/index.php/AAAI/article/view/8990.

[71] Supratik Chakraborty et al., « On Parallel Scalable Uniform SAT Witness Generation », *in*: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ed. by Christel Baier and Cesare Tinelli, vol. 9035, Lecture Notes in Computer Science, Springer, 2015, pp. 304–319, URL: https://doi.org/10.1007/978-3-662-46681-0_25.

[72] Rina Dechter et al., « Generating Random Solutions for Constraint Satisfaction Problems », *in*: *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, ed. by Rina Dechter, Michael J. Kearns, and Richard S. Sutton, AAAI Press / The MIT Press, 2002, pp. 15–21, URL: https://cdn.aaai.org/AAAI/2002/AAAI02-003.pdf.

[73] Rafael Dutra et al., « Efficient sampling of SAT solutions for testing », *in*: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, ed. by Michel Chaudron et al., ACM, 2018, pp. 549–559, URL: https://doi.org/10.1145/3180155.3180248.

[74] Stefano Ermon, Carla P. Gomes, and Bart Selman, « Uniform Solution Sampling Using a Constraint Solver As an Oracle », *in*: *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*,

ed. by Nando de Freitas and Kevin P. Murphy, AUAI Press, 2012, pp. 255–264, URL: https://doi.org/10.48550/arXiv.1210.4861.

[75] Stefano Ermon et al., « Embed and Project: Discrete Sampling with Universal Hashing », *in*: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, ed. by Christopher J. C. Burges et al., 2013, pp. 2085–2093, URL: https://proceedings.neurips.cc/paper/2013/hash/6d70cb65d15211726dcce4c0e971e21c-Abstract.html.

[76] Vibhav Gogate and Rina Dechter, « A New Algorithm for Sampling CSP Solutions Uniformly at Random », *in*: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, ed. by Frédéric Benhamou, vol. 4204, Lecture Notes in Computer Science, Springer, 2006, pp. 711–715, URL: https://doi.org/10.1007/11889205_56.

[77] Vibhav Gogate and Rina Dechter, « Studies in Solution Sampling », *in*: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, ed. by Dieter Fox and Carla P. Gomes, AAAI Press, 2008, pp. 271–276, URL: https://cdn.aaai.org/AAAI/2008/AAAI08-043.pdf.

[78] Priyanka Golia et al., « Designing Samplers is Easy: The Boon of Testers », *in*: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, IEEE, 2021, pp. 222–230, URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_31.

[79] Carla P. Gomes, Ashish Sabharwal, and Bart Selman, « Near-Uniform Sampling of Combinatorial Spaces Using XOR Constraints », *in*: *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, ed. by Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, MIT Press, 2006, pp. 481–488, URL: https://proceedings.neurips.cc/paper/2006/hash/4110a1994471c595f7583ef1b74ba4cb-Abstract.html.

[80] Rahul Gupta et al., « WAPS: Weighted and Projected Sampling », *in*: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, ed. by Tomás Vojnar and Lijun Zhang, vol. 11427, Lecture Notes in Computer Science, Springer, 2019, pp. 59–76, URL: https://doi.org/10.1007/978-3-030-17462-0_4.

[81] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani, « Random Generation of Combinatorial Structures from a Uniform Distribution », *in*: *Theor. Comput. Sci.* 43 (1986), pp. 169–188, URL: https://doi.org/10.1016/0304-3975(86)90174-X.

[82] Kuldeep S. Meel, « Constrained Counting and Sampling: Bridging the Gap between Theory and Practice », *in*: *CoRR* abs/1806.02239 (2018), URL: http://arxiv.org/abs/1806.02239.

[83] Kuldeep S. Meel, Yash Pote, and Sourav Chakraborty, « On Testing of Samplers », *in*: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, ed. by Hugo Larochelle et al., 2020, URL: https://proceedings.neurips.cc/paper/2020/hash/3f1656d9668dffcf8119e3ecff873558-Abstract.html.

[84] Jeho Oh et al., « Uniform sampling from kconfig feature models », *in*: *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19* 2 (2019).

[85] Guillaume Perez and Jean-Charles Régin, « MDDs: Sampling and Probability Constraints », *in*: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, ed. by J. Christopher Beck, vol. 10416, Lecture Notes in Computer Science, Springer, 2017, pp. 226–242, URL: https://doi.org/10.1007/978-3-319-66158-2_15.

[86] Gilles Pesant, Claude-Guy Quimper, and Hélène Verhaeghe, « Practically Uniform Solution Sampling in Constraint Programming », *in*: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, ed. by Pierre Schaus, vol. 13292, Lecture Notes in Computer Science, Springer, 2022, pp. 335–344, URL: https://doi.org/10.1007/978-3-031-08011-1_22.

[87] Bart Selman, Henry A. Kautz, and Bram Cohen, « Local search strategies for satisfiability testing », *in*: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, ed. by David S. Johnson and Michael A. Trick, vol. 26, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, DIMACS/AMS, 1993, pp. 521–531, URL: https://doi.org/10.1090/dimacs/026/25.

[88] Shubham Sharma et al., « Knowledge Compilation meets Uniform Sampling », *in*: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, ed. by Gilles Barthe, Geoff Sutcliffe,

and Margus Veanes, vol. 57, EPiC Series in Computing, EasyChair, 2018, pp. 620–636, URL: https://doi.org/10.29007/h4p9.

[89] Mate Soos, Stephan Gocht, and Kuldeep S. Meel, « Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling », *in*: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ed. by Shuvendu K. Lahiri and Chao Wang, vol. 12224, Lecture Notes in Computer Science, Springer, 2020, pp. 463–484, URL: https://doi.org/10.1007/978-3-030-53288-8_22.

[90] Mate Soos et al., « On Quantitative Testing of Samplers », *in*: *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, ed. by Christine Solnon, vol. 235, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 36:1–36:16, URL: https://doi.org/10.4230/LIPIcs.CP.2022.36.

[91] Marc Thurley, « sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP », *in*: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, ed. by Armin Biere and Carla P. Gomes, vol. 4121, Lecture Notes in Computer Science, Springer, 2006, pp. 424–429, URL: https://doi.org/10.1007/11814948_38.

[92] Leslie G. Valiant and Vijay V. Vazirani, « NP Is as Easy as Detecting Unique Solutions », *in*: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, ed. by Robert Sedgewick, ACM, 1985, pp. 458–463, URL: https://doi.org/10.1145/22145.22196.

[93] Wei Wei, Jordan Erenrich, and Bart Selman, « Towards Efficient Sampling: Exploiting Random Walk Strategies », *in*: *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, ed. by Deborah L. McGuinness and George Ferguson, AAAI Press / The MIT Press, 2004, pp. 670–676, URL: https://cdn.aaai.org/AAAI/2004/AAAI04-106.pdf.

[94] Yongjie Xu, Fu Song, and Taolue Chen, « ESampler: Boosting sampling of satisfying assignments for Boolean formulas via derivation », *in*: *J. Syst. Archit.* 129 (2022), p. 102615, URL: https://doi.org/10.1016/j.sysarc.2022.102615.

[95] Jun Yuan et al., « Modeling design constraints and biasing in simulation using BDDs », *in*: *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design, 1999, San Jose, California, USA, November 7-11, 1999*, ed. by Jacob K. White

and Ellen Sentovich, IEEE Computer Society, 1999, pp. 584–590, URL: https://doi.org/10.1109/ICCAD.1999.810715.

Pattern Mining

[96] Rakesh Agrawal and Ramakrishnan Srikant, « Fast Algorithms for Mining Association Rules in Large Databases », *in*: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, ed. by Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, Morgan Kaufmann, 1994, pp. 487–499, URL: http://www.vldb.org/conf/1994/P487.PDF.

[97] Mohamed-Bachir Belaid, Christian Bessiere, and Nadjib Lazaar, « Constraint Programming for Mining Borders of Frequent Itemsets », *in*: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, ed. by Sarit Kraus, ijcai.org, 2019, pp. 1064–1070, URL: https://doi.org/10.24963/ijcai.2019/149.

[98] Adnene Belfodil et al., « FSSD - A Fast and Efficient Algorithm for Subgroup Set Discovery », *in*: *2019 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2019, Washington, DC, USA, October 5-8, 2019*, ed. by Lisa Singh et al., IEEE, 2019, pp. 91–99, URL: https://doi.org/10.1109/DSAA.2019.00023.

[99] Anes Bendimerad et al., « Gibbs Sampling Subjectively Interesting Tiles », *in*: *Advances in Intelligent Data Analysis XVIII - 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27-29, 2020, Proceedings*, ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl, vol. 12080, Lecture Notes in Computer Science, Springer, 2020, pp. 80–92, URL: https://doi.org/10.1007/978-3-030-44584-3_7.

[100] Tijl De Bie, « Maximum entropy models and subjective interestingness: an application to tiles in binary databases », *in*: *Data Min. Knowl. Discov.* 23.*3* (2011), pp. 407–446, URL: https://doi.org/10.1007/s10618-010-0209-3.

[101] Mario Boley, Sandy Moens, and Thomas Gärtner, « Linear space direct pattern sampling using coupling from the past », *in*: *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, ed. by Qiang Yang, Deepak Agarwal, and Jian Pei, ACM, 2012, pp. 69–77, URL: https://doi.org/10.1145/2339530.2339545.

[102] Francesco Bonchi and Claudio Lucchese, « On Closed Constrained Frequent Pattern Mining », *in*: *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*, IEEE Computer Society, 2004, pp. 35–42, URL: https://doi.org/10.1109/ICDM.2004.10093.

[103] Guillaume Bosc et al., « Anytime discovery of a diverse set of patterns with Monte Carlo tree search », *in*: *Data Min. Knowl. Discov.* 32.*3* (2018), pp. 604–650, URL: https://doi.org/10.1007/s10618-017-0547-5.

[104] Abdelhamid Boudane et al., « Enumerating Non-redundant Association Rules Using Satisfiability », *in*: *Advances in Knowledge Discovery and Data Mining - 21st Pacific-Asia Conference, PAKDD 2017, Jeju, South Korea, May 23-26, 2017, Proceedings, Part I*, ed. by Jinho Kim et al., vol. 10234, Lecture Notes in Computer Science, 2017, pp. 824–836, URL: https://doi.org/10.1007/978-3-319-57454-7_64.

[105] Björn Bringmann and Albrecht Zimmermann, « The Chosen Few: On Identifying Valuable Patterns », *in*: *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA*, IEEE Computer Society, 2007, pp. 63–72, URL: https://doi.org/10.1109/ICDM.2007.85.

[106] Toon Calders, Christophe Rigotti, and Jean-François Boulicaut, « A Survey on Condensed Representations for Frequent Sets », *in*: *Constraint-Based Mining and Inductive Databases, European Workshop on Inductive Databases and Constraint Based Mining, Hinterzarten, Germany, March 11-13, 2004, Revised Selected Papers*, ed. by Jean-François Boulicaut, Luc De Raedt, and Heikki Mannila, vol. 3848, Lecture Notes in Computer Science, Springer, 2004, pp. 64–80, URL: https://doi.org/10.1007/11615576_4.

[107] Vladimir Dzyuba, Matthijs van Leeuwen, and Luc De Raedt, « Flexible constrained sampling with guarantees for pattern mining », *in*: *Data Min. Knowl. Discov.* 31.*5* (2017), pp. 1266–1293, URL: https://doi.org/10.1007/s10618-017-0501-6.

[108] Tias Guns, Siegfried Nijssen, and Luc De Raedt, « Itemset mining: A constraint programming perspective », *in*: *Artif. Intell.* 175.*12-13* (2011), pp. 1951–1983, URL: https://doi.org/10.1016/j.artint.2011.05.002.

[109] Arnold Hien et al., « A Relaxation-Based Approach for Mining Diverse Closed Patterns », *in*: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part I*, ed. by Frank Hutter et al., vol. 12457, Lecture Notes in Computer Science, Springer, 2020, pp. 36–54, URL: https://doi.org/10.1007/978-3-030-67658-2_3.

[110] Amina Kemmar et al., « Prefix-projection global constraint and top-k approach for sequential pattern mining », *in*: *Constraints An Int. J.* 22.*2* (2017), pp. 265–306, URL: https://doi.org/10.1007/s10601-016-9252-z.

[111] Arno J. Knobbe and Eric K. Y. Ho, « Maximally informative k-itemsets and their efficient discovery », *in*: *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, ed. by Tina Eliassi-Rad et al., ACM, 2006, pp. 237–244, URL: https://doi.org/10.1145/1150402.1150431.

[112] Arno J. Knobbe and Eric K. Y. Ho, « Pattern Teams », *in*: *Knowledge Discovery in Databases: PKDD 2006, 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, Berlin, Germany, September 18-22, 2006, Proceedings*, ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, vol. 4213, Lecture Notes in Computer Science, Springer, 2006, pp. 577–584, URL: https://doi.org/10.1007/11871637_58.

[113] Gokberk Kocak et al., « Closed Frequent Itemset Mining with Arbitrary Side Constraints », *in*: *2018 IEEE International Conference on Data Mining Workshops, ICDM Workshops, Singapore, Singapore, November 17-20, 2018*, ed. by Hanghang Tong et al., IEEE, 2018, pp. 1224–1232, URL: https://doi.org/10.1109/ICDMW.2018.00175.

[114] Gökberk Koçak et al., « Exploiting Incomparability in Solution Dominance: Improving General Purpose Constraint-Based Mining », *in*: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, ed. by Giuseppe De Giacomo et al., vol. 325, Frontiers in Artificial Intelligence and Applications, IOS Press, 2020, pp. 331–338, URL: https://doi.org/10.3233/FAIA200110.

[115] Nadjib Lazaar et al., « A Global Constraint for Closed Frequent Pattern Mining », *in*: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, ed. by Michel Rueher, vol. 9892, Lecture Notes in Computer Science, Springer, 2016, pp. 333–349, URL: https://doi.org/10.1007/978-3-319-44953-1_22.

[116] Benjamin Négrevergne et al., « Dominance Programming for Itemset Mining », *in*: *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, ed. by Hui Xiong et al., IEEE Computer Society, 2013, pp. 557–566, URL: https://doi.org/10.1109/ICDM.2013.92.

[117] Nicolas Pasquier et al., « Discovering Frequent Closed Itemsets for Association Rules », *in*: *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, ed. by Catriel Beeri and Peter Buneman, vol. 1540, Lecture Notes in Computer Science, Springer, 1999, pp. 398–416, URL: https://doi.org/10.1007/3-540-49257-7_25.

[118] Luc De Raedt and Albrecht Zimmermann, « Constraint-Based Pattern Set Mining », *in*: *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*, SIAM, 2007, pp. 237–248, URL: https://doi.org/10.1137/1.9781611972771.22.

[119] Pierre Schaus, John O. R. Aoga, and Tias Guns, « CoverSize: A Global Constraint for Frequency-Based Itemset Mining », *in*: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, ed. by J. Christopher Beck, vol. 10416, Lecture Notes in Computer Science, Springer, 2017, pp. 529–546, URL: https://doi.org/10.1007/978-3-319-66158-2_34.

[120] Jianyong Wang et al., « TFP: An Efficient Algorithm for Mining Top-K Frequent Closed Itemsets », *in*: *IEEE Trans. Knowl. Data Eng.* 17.*5* (2005), pp. 652–664, URL: https://doi.org/10.1109/TKDE.2005.81.

Feature Models

[121] Mathieu Acher et al., « VaryLATEX: Learning Paper Variants That Meet Constraints », *in*: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, ed. by Rafael Capilla, Malte Lochau, and Lidia Fuentes, ACM, 2018, pp. 83–88, URL: https://doi.org/10.1145/3168365.3168372.

[122] Mustafa Al-Hajjaji et al., « IncLing: efficient product-line testing using incremental pairwise sampling », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, ed. by Bernd Fischer and Ina Schaefer, ACM, 2016, pp. 144–155, URL: https://doi.org/10.1145/2993236.2993253.

[123] Eduard Baranov, Axel Legay, and Kuldeep S. Meel, « Baital: an adaptive weighted sampling approach for improved t-wise coverage », *in*: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, ed. by Prem Devanbu,

Myra B. Cohen, and Thomas Zimmermann, ACM, 2020, pp. 1114–1126, URL: https://doi.org/10.1145/3368089.3409744.

[124] Eduard Baranov et al., « A Scalable t-wise Coverage Estimator », *in*: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, pp. 36–47, URL: https://doi.org/10.1145/3510003.3510218.

[125] Don S. Batory, « Feature Models, Grammars, and Propositional Formulas », *in*: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, ed. by J. Henk Obbink and Klaus Pohl, vol. 3714, Lecture Notes in Computer Science, Springer, 2005, pp. 7–20, URL: https://doi.org/10.1007/11554844_3.

[126] David Benavides, Sergio Segura, and Antonio Ruiz Cortés, « Automated analysis of feature models 20 years later: A literature review », *in*: *Inf. Syst.* 35.*6* (2010), pp. 615–636, URL: https://doi.org/10.1016/j.is.2010.01.001.

[127] Anastasia Cmyrev and Ralf Reissing, « Efficient and effective testing of automotive software product lines », *in*: *Applied Science and Engineering Progress* 7.*2* (2014), pp. 53–57, URL: http://doi.org/10.14416/j.ijast.2014.05.001.

[128] David M. Cohen et al., « The AETG system: An approach to testing based on combinatorial design », *in*: *IEEE Transactions on Software Engineering* 23.*7* (1997), pp. 437–444, URL: https://doi.org/10.1109/32.605761.

[129] David Fernández-Amorós et al., « A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models », *in*: *IEEE Trans. Software Eng.* 40.*9* (2014), pp. 895–910, URL: https://doi.org/10.1109/TSE.2014.2331073.

[130] Axel Halin et al., « Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack », *in*: *Empir. Softw. Eng.* 24.*2* (2019), pp. 674–717, URL: https://doi.org/10.1007/s10664-018-9635-4.

[131] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb, « PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models », *in*: *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, ed. by Tadashi Dohi and Bojan Cukic, IEEE Computer Society, 2011, pp. 120–129, URL: https://doi.org/10.1109/ISSRE.2011.31.

[132] Aymeric Hervieu et al., « Practical minimization of pairwise-covering test configurations using constraint programming », *in*: *Inf. Softw. Technol.* 71 (2016), pp. 129–146, URL: https://doi.org/10.1016/j.infsof.2015.11.007.

[133] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey, « An algorithm for generating t-wise covering arrays from large feature models », *in*: *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, ed. by Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides, ACM, 2012, pp. 46–55, URL: https://doi.org/10.1145/2362536.2362547.

[134] Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru, « From extended feature models to constraint logic programming », *in*: *Sci. Comput. Program.* 78.*12* (2013), pp. 2295–2312, URL: https://doi.org/10.1016/j.scico.2012.06.004.

[135] Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru, « Global Constraints on Feature Models », *in*: *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, ed. by David Cohen, vol. 6308, Lecture Notes in Computer Science, Springer, 2010, pp. 537–551, URL: https://doi.org/10.1007/978-3-642-15396-9_43.

[136] Alexander Knüppel et al., « Is there a mismatch between real-world feature models and product-line research? », *in*: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, ed. by Eric Bodden et al., ACM, 2017, pp. 291–302, URL: https://doi.org/10.1145/3106237.3106252.

[137] D Richard Kuhn, Raghu N Kacker, and Yu Lei, *Introduction to combinatorial testing*, CRC press, 2013, URL: https://doi.org/10.1007/978-3-662-43429-1_1.

[138] David Kuhn, Raghu Kacker, and Yu Lei, *Practical Combinatorial Testing*, en, July 2010, URL: https://doi.org/10.6028/NIST.SP.800-142.

[139] Thomas von der Maßen and Horst Lichter, « Determining the Variation Degree of Feature Models », *in*: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, ed. by J. Henk Obbink and Klaus Pohl, vol. 3714, Lecture Notes in Computer Science, Springer, 2005, pp. 82–88, URL: https://doi.org/10.1007/11554844_9.

[140] Jean Melo et al., « A Quantitative Analysis of Variability Warnings in Linux », *in*: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016*, ed. by Ina Schaefer, Vander Alves, and Eduardo Santana de Almeida, ACM, 2016, pp. 3–8, URL: https://doi.org/10.1145/2866614.2866615.

[141] Jeho Oh, Paul Gazzillo, and Don S. Batory, « t-wise coverage by uniform sampling », *in*: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, ed. by Thorsten Berger et al., ACM, 2019, 15:1–15:4, URL: https://doi.org/10.1145/3336294.3342359.

[142] Chico Sundermann et al., « Yet another textual variability language?: a community effort towards a unified language », *in*: *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, ed. by Mohammad Reza Mousavi and Pierre-Yves Schobbens, ACM, 2021, pp. 136–147, URL: https://doi.org/10.1145/3461001.3471145.

[143] Akihisa Yamada et al., « Greedy combinatorial test case generation using unsatisfiable cores », *in*: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, ed. by David Lo, Sven Apel, and Sarfraz Khurshid, ACM, 2016, pp. 614–624, URL: https://doi.org/10.1145/2970276.2970335.

Multi-objective

[144] Charles Audet et al., « Performance indicators in multiobjective optimization », *in*: *Eur. J. Oper. Res.* 292.*2* (2021), pp. 397–422, URL: https://doi.org/10.1016/j.ejor.2020.11.016.

[145] Anne Auger et al., « Investigating and exploiting the bias of the weighted hypervolume to articulate user preferences », *in*: *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, ed. by Franz Rothlauf, ACM, 2009, pp. 563–570, URL: https://doi.org/10.1145/1569901.1569980.

[146] Nicola Beume et al., « On the Complexity of Computing the Hypervolume Indicator », *in*: *IEEE Trans. Evol. Comput.* 13.*5* (2009), pp. 1075–1082, URL: https://doi.org/10.1109/TEVC.2009.2015575.

[147] Karl Bringmann, Tobias Friedrich, and Patrick Klitzke, « Two-dimensional subset selection for hypervolume and epsilon-indicator », *in*: *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014*, ed. by Dirk V. Arnold, ACM, 2014, pp. 589–596, URL: https://doi.org/10.1145/2576768.2598276.

[148] Indraneel Das and John E. Dennis, « Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems », *in*: *SIAM J. Optim.* 8.*3* (1998), pp. 631–657, URL: https://doi.org/10.1137/S1052623496307510.

[149] Kalyanmoy Deb, Sunith Bandaru, and Haitham Seada, « Generating Uniformly Distributed Points on a Unit Simplex for Evolutionary Many-Objective Optimization », *in*: *Evolutionary Multi-Criterion Optimization - 10th International Conference, EMO 2019, East Lansing, MI, USA, March 10-13, 2019, Proceedings*, ed. by Kalyanmoy Deb et al., vol. 11411, Lecture Notes in Computer Science, Springer, 2019, pp. 179–190, URL: https://doi.org/10.1007/978-3-030-12598-1_15.

[150] Kalyanmoy Deb and Himanshu Jain, « An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints », *in*: *IEEE Trans. Evol. Comput.* 18.*4* (2014), pp. 577–601, URL: https://doi.org/10.1109/TEVC.2013.2281535.

[151] Emir Demirovic and Nicolas Schwind, « Representative Solutions for Bi-Objective Optimisation », *in*: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, AAAI Press, 2020, pp. 1436–1443, URL: https://ojs.aaai.org/index.php/AAAI/article/view/5501.

[152] Renaud Hartert and Pierre Schaus, « A Support-Based Algorithm for the Bi-Objective Pareto Constraint », *in*: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, ed. by Carla E. Brodley and Peter Stone, Québec, Canada: AAAI Press, 2014, pp. 2674–2679, URL: https://cdn.aaai.org/ojs/9119/9119-13-12647-1-2-20201228.pdf.

[153] Tobias Kuhn et al., « Hypervolume Subset Selection in Two Dimensions: Formulations and Algorithms », *in*: *Evol. Comput.* 24.*3* (2016), pp. 411–425, URL: https://doi.org/10.1162/EVCO_a_00157.

[154] Miqing Li and Xin Yao, « Quality Evaluation of Solution Sets in Multiobjective Optimisation: A Survey », *in*: *ACM Comput. Surv.* 52.*2* (2019), 26:1–26:38, URL: https://doi.org/10.1145/3300148.

[155] William Reed, « Random points in a simplex », *in*: *Pacific Journal of Mathematics* 54.*2* (1974), pp. 183–198, URL: http://doi.org/10.2140/pjm.1974.54.183.

[156] Pierre Schaus and Renaud Hartert, « Multi-Objective Large Neighborhood Search », *in*: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, ed. by Christian Schulte, vol. 8124, Lecture Notes in Computer Science, Springer, 2013, pp. 611–627, URL: https://doi.org/10.1007/978-3-642-40627-0_46.

[157] Nicolas Schwind et al., « Representative Solutions for Multi-Objective Constraint Optimization Problems », *in*: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, ed. by Chitta Baral, James P. Delgrande, and Frank Wolter, AAAI Press, 2016, pp. 601–604, URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12873.

[158] Charles Vernerey, personal communication, Feb. 3, 2023.

[159] Eckart Zitzler et al., « Performance assessment of multiobjective optimizers: an analysis and review », *in*: *IEEE Trans. Evol. Comput.* 7.*2* (2003), pp. 117–132, URL: https://doi.org/10.1109/TEVC.2003.810758.

Other References

[160] George Lowther (https://mathoverflow.net/users/1004/george-lowther) (version: 2016-08-08), *Mean minimum distance for N random points on a one-dimensional line*, MathOverflow, URL: https://mathoverflow.net/q/1308.

[161] Henry (https://math.stackexchange.com/users/6460/henry) (version: 2017-04-13), *How is the distance of two random points in a unit hypercube distributed?*, Mathematics Stack Exchange, URL: https://math.stackexchange.com/q/1985698.

[162] Franz Aurenhammer and Rolf Klein, « Voronoi Diagrams », *in*: *Handbook of Computational Geometry*, ed. by Jörg-Rüdiger Sack and Jorge Urrutia, North Holland / Elsevier, 2000, pp. 201–290, URL: https://doi.org/10.1016/b978-044482537-7/50006-1.

[163] Eric Bach, « Realistic Analysis of Some Randomized Algorithms », *in*: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, ed. by Alfred V. Aho, ACM, 1987, pp. 453–461, URL: https://doi.org/10.1145/28395.28444.

[164] Tugkan Batu et al., « Testing Closeness of Discrete Distributions », *in*: *J. ACM* 60.*1* (2013), 4:1–4:25, URL: https://doi.org/10.1145/2432622.2432626.

[165] Felix Brandt et al., eds., *Handbook of Computational Social Choice*, Cambridge University Press, 2016, ISBN: 9781107446984, URL: https://doi.org/10.1017/CBO9781107446984.

[166] Yann Chevaleyre et al., « Issues in Multiagent Resource Allocation », *in*: *Informatica (Slovenia)* 30.*1* (2006), pp. 3–31, URL: http://www.informatica.si/index.php/informatica/article/view/70.

[167] Stephen A. Cook, « The Complexity of Theorem-Proving Procedures », *in*: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, ACM, 1971, pp. 151–158, URL: https://doi.org/10.1145/800157.805047.

[168] Rina Dechter, Kalev Kask, and Robert Mateescu, « Iterative Join-Graph Propagation », *in*: *UAI '02, Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence, University of Alberta, Edmonton, Alberta, Canada, August 1-4, 2002*, ed. by Adnan Darwiche and Nir Friedman, Morgan Kaufmann, 2002, pp. 128–136, URL: https://doi.org/10.48550/arXiv.1301.0564.

[169] Prafulla Dhariwal et al., « Jukebox: A Generative Model for Music », *in*: *CoRR* abs/2005.00341 (2020), URL: https://arxiv.org/abs/2005.00341.

[170] R. A. Fisher and L. H. C. Tippett, « Limiting forms of the frequency distribution of the largest or smallest member of a sample », *in*: *Mathematical Proceedings of the Cambridge Philosophical Society* 24.*2* (1928), pp. 180–190, URL: https://doi.org/10.1017/S0305004100015681.

[171] Peter A Henderson, *Practical methods in ecology*, John Wiley & Sons, 2003, ISBN: 978-1-444-31227-0.

[172] Caleb H. Johnson et al., « Evaluation of Algorithms for Randomizing Key Item Locations in Game Worlds », *in*: *IEEE Access* 9 (2021), pp. 48286–48302, URL: https://doi.org/10.1109/ACCESS.2021.3069114.

[173] Richard M. Karp, « Reducibility Among Combinatorial Problems », *in*: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, ed. by Raymond E. Miller and James W. Thatcher, The IBM Research Symposia Series, Plenum Press, New York, 1972, pp. 85–103, URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[174] Nathan Kitchen and Andreas Kuehlmann, « Stimulus generation for constrained random simulation », *in*: *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, ed. by Georges G. E. Gielen, IEEE Computer Society, 2007, pp. 258–265, URL: https://doi.org/10.1109/ICCAD.2007.4397275.

[175] Donald E Knuth, *Art of computer programming, volume 2: Seminumerical algorithms*, Addison-Wesley Professional, 2014, ISBN: 978-0-201-89684-8.

[176] Jean-Marie Lagniez and Pierre Marquis, « An Improved Decision-DNNF Compiler », *in*: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ed. by Carles Sierra, ijcai.org, 2017, pp. 667–673, URL: https://doi.org/10.24963/ijcai.2017/93.

[177] Stuart P. Lloyd, « Least squares quantization in PCM », *in*: *IEEE Trans. Inf. Theory* 28.*2* (1982), pp. 129–136, URL: https://doi.org/10.1109/TIT.1982.1056489.

[178] Michael Luby, Alistair Sinclair, and David Zuckerman, « Optimal Speedup of Las Vegas Algorithms », *in*: *Inf. Process. Lett.* 47.*4* (1993), pp. 173–180, URL: https://doi.org/10.1016/0020-0190(93)90029-9.

[179] Christian J. Muise et al., « Dsharp: Fast d-DNNF Compilation with sharpSAT », *in*: *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, ed. by Leila Kosseim and Diana Inkpen, vol. 7310, Lecture Notes in Computer Science, Springer, 2012, pp. 356–361, URL: https://doi.org/10.1007/978-3-642-30353-1_36.

[180] Christos H. Papadimitriou, « On Selecting a Satisfying Truth Assignment (Extended Abstract) », *in*: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, IEEE Computer Society, 1991, pp. 163–169, URL: https://doi.org/10.1109/SFCS.1991.185365.

[181] Karl Pearson, « X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling », *in*: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.*302* (1900), pp. 157–175, URL: https://doi.org/10.1007/978-1-4612-4380-9_2.

[182] Stephen Plaza, Igor L. Markov, and Valeria Bertacco, « Random Stimulus Generation using Entropy and XOR Constraints », *in*: *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, ed. by Donatella Sciuto, ACM, 2008, pp. 664–669, URL: https://doi.org/10.1109/DATE.2008.4484754.

[183] Quentin Plazar et al., « Efficient and Complete FD-solving for extended array constraints », *in*: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ed. by Carles Sierra, ijcai.org, 2017, pp. 1231–1238, URL: https://doi.org/10.24963/ijcai.2017/171.

[184] Björn Regnell and Krzysztof Kuchcinski, « Exploring Software Product Management decision problems with constraint solving - opportunities for prioritization and release planning », *in*: *5th International Workshop on Software Product Management, IWSPM 2011, Trento, Italy, August 30, 2011*, ed. by Marjo Kauppinen and Krzysztof Wnuk, IEEE Computer Society, 2011, pp. 47–56, URL: https://doi.org/10.1109/IWSPM.2011.6046203.

[185] Christian P. Robert and George Casella, *Monte Carlo Statistical Methods*, Springer Texts in Statistics, Springer, 2004, ISBN: 978-1-4419-1939-7, URL: https://doi.org/10.1007/978-1-4757-4145-2.

[186] Günther Ruhe and Moshood Omolade Saliu, « The Art and Science of Software Release Planning », *in*: *IEEE Softw.* 22.*6* (2005), pp. 47–53, URL: https://doi.org/10.1109/MS.2005.164.

[187] Luis A. Santaló and Mark Kac, *Integral Geometry and Geometric Probability*, 2nd ed., Cambridge Mathematical Library, Cambridge University Press, 2004, ISBN: 9780511617331, URL: https://doi.org/10.1017/CBO9780511617331.

[188] Uwe Schöning, « A Probabilistic Algorithm for k-SAT and Constraint Satisfaction Problems », *in*: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, IEEE Computer Society, 1999, pp. 410–414, URL: https://doi.org/10.1109/SFFCS.1999.814612.

[189] Helmut Simonis, « Kakuro as a constraint problem », *in*: *Proc. seventh Int. Works. on Constraint Modelling and Reformulation* (2008).

[190] Kate Smith-Miles and Simon Bowly, « Generating new test instances by evolving in instance space », *in*: *Comput. Oper. Res.* 63 (2015), pp. 102–113, URL: https://doi.org/10.1016/j.cor.2015.04.022.

[191] Larry J. Stockmeyer, « On Approximation Algorithms for #P », *in*: *SIAM J. Comput.* 14.*4* (1985), pp. 849–861, URL: https://doi.org/10.1137/0214060.

[192] Alastair J. Walker, « An Efficient Method for Generating Discrete Random Variables with General Distributions », *in*: *ACM Trans. Math. Softw.* 3.*3* (1977), pp. 253–256, URL: https://doi.org/10.1145/355744.355749.

Part V

Appendices

Appendix A

Proofs

A.1 RandomSearch's Distribution

This section proves Theorem 5. The proof done is in two steps. First, we need to define some mathematical objects and show some properties of them. We then link these objects to the probability of RandomSearch to sample a given combination.

First, we introduce a family of polynomials that appear in the lower bound we are proving.

Definition 49. We note P^t the only polynomial of degree $t-1$ such that $P^t(i) = t! \cdot 2^i$ for i in $\{0, \dots, t-1\}$. By convention, we extend the definition to $P^0(n) = 0$.

There is a unique polynomial of degree d which passes through $d+1$ points, so P^t is well defined. We now prove a recurrence formula for these polynomials.

Lemma 4. *The polynomials P^t follow the recurrence relation*

$$P^t(n) = P^t(n-1) + t \cdot P^{t-1}(n-1)\,.$$

Proof. Let $P = \frac{P^{t+1}(n+1) - P^{t+1}(n)}{t+1}$. We want to prove that $P = P^t$, so we need to show that it has degree $t-1$ and that for $i \in \{0, \dots, t-1\}, P(i) = P^t = t! \cdot 2^i$.

- The monomial of degree t cancels in the substraction $P^{t+1}(n+1) - P^{t+1}(n)$, so P only has degree $t-1$
- Let $i \in \{0, \dots t-1\}$,

$$\begin{aligned} P(i) &= \frac{P^{t+1}(i+1) - P^{t+1}(i)}{t+1} \\ &= \frac{(t+1)! \cdot 2^{i+1} - (t+1)! \cdot 2^i}{t+1} \\ &= t! \cdot 2^i \end{aligned}$$

So $P = P^t$. The property follows from a rearrangement of the equality. □

With these polynomials, we can define a (two dimensional) sequence. This sequence is the lower bound of $p_\sigma^{\mathcal{R}}$, which is used in the theorem we want to prove.

Definition 50. We define the two dimensional sequence u on $0 \leq t \leq n$ as

$$u_n^t = \frac{(n-t)! \cdot (t! \cdot 2^n - P^t(n))}{n! \cdot 2^n}.$$

This sequence is used as a lower bound in the theorem we want to prove. We first show a convergence property of u.

Lemma 5. *Let t be an integer, then u_n^t is equivalent to $1/\binom{n}{t}$ when n tends to infinity, i.e.*

$$u_n^t \underset{n\to\infty}{\sim} 1/\binom{n}{t}.$$

Proof.

$$\begin{aligned} u_n^t \cdot \binom{n}{t} &= \frac{(n-t)! \cdot (t! \cdot 2^n - P^t(n))}{n! \cdot 2^n} \cdot \binom{n}{t} \\ &= \frac{(t! \cdot 2^n - P^t(n))}{t! \cdot 2^n} \\ &= 1 - \frac{P^t(n)}{t! \cdot 2^n} \\ &\xrightarrow[n\to\infty]{} 1 \end{aligned}$$

□

To link this sequence to the probability we are studying, we use the following lemma, which gives a recurrence relation for u_n^t.

Lemma 6. *For n and t integers such that $1 \leq t \leq n$, the sequence u follows the recurrence relation*

$$u_n^t = \frac{1}{2n}\left(t \cdot u_{n-1}^{t-1} + (n-t) \cdot u_{n-1}^t\right)$$

Proof. We simply replace the definition of u in the right hand side of the equality.

$$\begin{aligned}
\frac{1}{2n}\left(t \cdot u_{n-1}^{t-1} + (n-t) \cdot u_{n-1}^{t}\right) &= \frac{1}{2n}\Bigg(t \cdot \frac{(n-t)! \cdot ((t-1)! \cdot 2^{n-1} - P^{t-1}(n-1))}{(n-1)! \cdot 2^{n-1}} \\
&\qquad + (n-t) \cdot \frac{(n-t-1)! \cdot (t! \cdot 2^{n-1} - P^{t}(n-1))}{(n-1)! \cdot 2^{n-1}}\Bigg) \\
&= \frac{(n-t)!}{n! \cdot 2^{n}}\left(t! \cdot 2^{n} - t \cdot P^{t-1}(n-1) - P^{t}(n-1)\right) \\
&= \frac{(n-t)! \cdot (t! \cdot 2^{n} - P^{t}(n))}{n! \cdot 2^{n}} \\
&= u_n^t
\end{aligned}$$

□

To prove the main theorem, we prove the following stronger version.

Lemma 7. *Let n be the number of features, and σ be a t-wise combination appearing on the set of solutions, then $p_\sigma \geq u_n^t$.*

Proof. We note l_n^t the lower bound, and we show that it verifies the same recurrence relation as u_n^t. We now consider a fixed t-wise combination σ, on a problem with n variables. We know that there is at least one configuration C containing σ. We recall that there is a Boolean variable X_f associated with each feature in $\mathcal{F}$. When all variables are instantiated, it defines a configuration such that $f \in C \Leftrightarrow X_f = 1$.

To obtain the recurrence relation, we apply the RandomSearch search strategy. Suppose that there are m uninstantiated variables. Then

- with probability $\frac{t}{m}$ one variable of σ is picked by RandomSearch (let's note it X_f). Then there is a $\frac{1}{2}$ probability that the correct value for X_f is chosen (i.e. $\sigma(X_f)$). Then the other $t-1$ variables must also be fixed, among the remaining $m-1$ uninstantiated variables. In the best case, some values are propagated, reducing the number of uninstantiated variables. In the worst case, there is no propagation. In this case, the probability of choosing the correct values for the $t-1$ remaining variables from the $m-1$ variables is l_{m-1}^{t-1}. Overall, this has a probability to happen of
$$\frac{t}{2m} l_{m-1}^{t-1}$$
- otherwise (so with probability $\frac{m-t}{m} = 1 - \frac{t}{m}$), a variable X_f is chosen that is not in σ. Then there is a chance that the value chosen for X_f does no longer allow

the combination σ. However, we know that there is (at least) one solution that contains σ, so if X_f takes the same value as in that solution, there is still a chance to return a solution with σ. Since there are two possible values for X_f, there is a $\frac{1}{2}$ chance that it takes the value in C (i.e. $X_f = 1 \Leftrightarrow f \in C$). Then, the t variables in σ still have to be chosen from the $m-1$ remaining variables, hence a probability of l^t_{m-1}.

Combining these probabilities, we have

$$p_\sigma \geq l^t_n = \frac{1}{2n}\left(t \cdot l^{t-1}_{n-1} + (n-t) \cdot l^t_{n-1}\right) .$$

l^t_n follows the same recurrence relation as u^t_n and has the same initial values ($l^0_n = 1$ for $n \geq 0$, and $l^t_t = 1/2^t$ for $t \geq 0$), so it is equal to u^t_n. □

We can now prove the main theorem simply by using the previous lemmas.

Theorem 5. *Given a feature model with n features, and σ an allowed t-wise combination, there is a sequence u^t_n such that*

$$p^{\mathcal{R}}_\sigma \geq u^t_n$$

and

$$u^t_n \underset{n\to\infty}{\sim} \frac{1}{\binom{n}{t}} .$$

Proof. We simply apply Lemmas 5 and 7. □

A.2 Sampling on Feature Diagrams

This section proves the theorems presented in Chapter 7 Section 7.6 about counting and sampling on feature diagrams.

A.2.1 Expansion Operator

To make the proofs easier, we first introduce an operator called the expansion.

Definition 51 (Expansion Operator)**.** Let E_1 and E_2 be two sets of configurations, we define the expansion operator as

$$E_1 \diamond E_2 = \bigcup_{\substack{C_1 \in E_1 \\ C_2 \in E_2}} \{C_1 \cup C_2\} .$$

Given $E = \{E_1, \ldots, E_n\}$ n sets of configurations, we extend the expansion operator to

$$\bigdiamond_{E_i \in E} E_i = E_1 \diamond \ldots \diamond E_n \,.$$

Remark. *As a consequence of the definition, an expansion on the empty set is:*

$$\bigdiamond_{E_i \in \emptyset} E_i = \{\emptyset\}$$

This definition is similar to the Cartesian product, but for merging sets of configurations. Informally, if there is a set of configurations E_1 on features $\mathcal{F}_1$, and E_2 on features $\mathcal{F}_2$ then $E_1 \diamond E_2$ is the allowed configurations on $\mathcal{F}_1 \cup \mathcal{F}_2$ (assuming there are no constraints between the features in $\mathcal{F}_1$ and $\mathcal{F}_2$).

This operator allows us to easily recursively compute the set of allowed configurations of a feature diagram.

Proposition 14 (Set of Configurations)**.** *The expansion operator can be used to recursively compute the set of allowed configurations of a feature diagram D:*

- *If* $D.\texttt{children} = \emptyset$*, then* $Sols\,(D) = \{\{D\}\}$
- *If* $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$*,*

$$Sols(D) = \{\{D.\texttt{feature}\}\} \diamond \bigdiamond_{D' \in D.\texttt{mand}} Sols(D') \diamond \bigdiamond_{D' \in D.\texttt{opt}} Sols(D') \cup \{\emptyset\}$$

- *If* $D.\texttt{xor} \neq \emptyset$*,*

$$Sols(D) = \{\{D.\texttt{feature}\}\} \diamond \bigcup_{D' \in D.\texttt{xor}} Sols(D')$$

- *If* $D.\texttt{or} \neq \emptyset$*,*

$$Sols(D) = \{\{D.\texttt{feature}\}\} \diamond \left(\left(\bigdiamond_{D' \in D.\texttt{or}} Sols(D') \cup \{\emptyset\}\right) \setminus \{\emptyset\}\right)$$

Proof. We recall that the expansion operator is the operator for merging sets of configurations. The formula boils down to 5 items:

- $\{\{D.\texttt{feature}\}\} \diamond \ldots$ is the part where the current feature is added to the set of

configurations;

- $\bigdiamond_{D' \in D.\texttt{mand}} Sols(D')$ is the part where all the configurations of all mandatory children are merged;
- $\bigdiamond_{D' \in D.\texttt{opt}} Sols(D') \cup \{\emptyset\}$ is the part for optional children. The singleton containing the empty set is a neutral element for the expansion operator. Adding the empty set to the set of configurations is a way to allowing of either allowing the children to take a configuration or not, which is exactly the definition of optional children;
- $\bigcup_{D' \in D.\texttt{xor}} Sols(D')$ simply does the union of the configuration of children, without the expansion operator because a single configuration is chosen from the *xor* children;
- $\left(\bigdiamond_{D' \in D.\texttt{or}} Sols(D') \cup \{\emptyset\}\right) \setminus \{\emptyset\}$ is almost the same as the optional children, except that at least one child must be chosen, so the empty set is removed.

□

A.2.2 Variation Degree

Before proving the formula for the variation degree, we show a lemma to show that it is easy to count with the expansion operator.

Lemma 8. *Let E_1 and E_2 be two sets of configurations on different sets of features. Then*

$$|E_1 \diamond E_2| = |E_1| \cdot |E_2|$$

Proof. If the sets of features of E_1 and E_2 are disjoint, then the union in the definition of the expansion operator is a disjoint union. Then

$$\begin{aligned} |E_1 \diamond E_2| &= \left| \bigcup_{\substack{C_1 \in E_1 \\ C_2 \in E_2}} \{C_1 \cup C_2\} \right| \\ &= \sum_{\substack{C_1 \in E_1 \\ C_2 \in E_2}} |\{C_1 \cup C_2\}| \\ &= \sum_{\substack{C_1 \in E_1 \\ C_2 \in E_2}} 1 \\ &= |E_1| \cdot |E_2| \end{aligned}$$

□

Theorem 6 (Variation Degree of Feature Diagrams [139])**.** *Let D be a feature diagram. Then*

- *If $D.\texttt{children} = \emptyset$, then $|Sols(D)| = 1$.*
- *If $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$,*

$$|Sols(D)| = \prod_{D' \in D.\texttt{mand}} |Sols(D')| \times \prod_{D' \in D.\texttt{opt}} |Sols(D')| + 1 .$$

- *If $D.\texttt{xor} \neq \emptyset$,*

$$|Sols(D)| = \sum_{D' \in D.\texttt{xor}} |Sols(D')| .$$

- *If $D.\texttt{or} \neq \emptyset$,*

$$|Sols(D)| = \left(\prod_{D' \in D.\texttt{or}} |Sols(D')| + 1 \right) - 1 .$$

Proof. The proof follows from Lemma 8 and Property 14. All the sub-feature diagrams use disjoint sets of features. □

A.2.3 Commonalities

Theorem 7 (Commonalities on Feature Diagrams [129])**.** *Let f be a feature and D be a feature diagram. We note $\phi_f(D) = |\{C \in Sols(D) \,|\, f \in C\}|$ the number of occurrences of a feature in the set of allowed configurations. Then*

$$\phi_f(D) = \begin{cases} |Sols(D)| & \text{if } D.\texttt{feature} = f \\ \frac{|Sols(D)|}{|Sols(D')|} \cdot \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{mand} \\ \frac{|Sols(D)|}{|Sols(D')|+1} \cdot \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{opt} \text{ or } D' \in D.\texttt{or} \\ \phi_f(D') & \text{if } f \in D' \text{ and } D' \in D.\texttt{xor} \end{cases}$$

The commonality of f in D can then be computed with $\varphi_f D = \frac{\phi_f(D)}{|Sols(D)|}$.

Proof. The idea of the proof is the same as for Theorem 6, but instead of computing the variation degree directly, we first restrict to the set of allowed configurations containing the selected feature.

Let D be a feature diagram and f a feature in it. We note $\Phi_f(D)$ the set of allowed configurations of D containing f (i.e. $\Phi_f(D) = \{C \in Sols(D) \,|\, f \in C\}$). Then $\phi_f(D) =$

$|\Phi_f(D)|$.

If $f = D.\texttt{feature}$, then all the allowed combinations of D contain f, so $\phi_f(D) = |Sols(D)|$. Now suppose that f is in some $D' \in D.\texttt{children}$. There are different cases depending on whether D' is in $D.\texttt{mand}$, $D.\texttt{opt}$, $D.\texttt{or}$ or $D.\texttt{xor}$:

- If $D' \in D.\texttt{mand}$, we split the formula of Property 14 between D' and the other children of D

$$\begin{aligned}
Sols(D) =& \{\{D.\texttt{feature}\}\} \diamond Sols(D') \diamond \bigdiamond_{D'' \in D.\texttt{mand} \setminus \{D'\}} Sols(D'') \\
\Phi_f(D) =& \{\{D.\texttt{feature}\}\} \diamond \Phi_f(D') \diamond \bigdiamond_{D'' \in D.\texttt{mand} \setminus \{D'\}} Sols(D'') \\
\phi_f(D) =& \phi_f(D') \times \prod_{D'' \in D.\texttt{mand} \setminus \{D'\}} |Sols(D'')| \\
\phi_f(D) =& \phi_f(D') \cdot \frac{|Sols(D)|}{|Sols(D')|}
\end{aligned}$$

- If $D' \in D.\texttt{opt}$ the same reasoning works, just by remarking that

$$\prod_{D'' \in D.\texttt{mand} \setminus \{D'\}} |Sols(D'')| = \frac{|Sols(D)|}{|Sols(D')| + 1}$$

  .
- If $D' \in D.\texttt{or}$, we remark that removing the empty set does not matter because we are interested in the solutions that *contain* the feature f. The formula of Property 14 for the $D.\texttt{or}$ children becomes the same as for the $D.\texttt{opt}$ children.
- If $D' \in D.\texttt{xor}$,

$$\begin{aligned}
Sols(D) =& \{\{D.\texttt{feature}\}\} \diamond \left(Sols(D') \cup \bigcup_{D'' \in D.\texttt{xor} \setminus \{D'\}} Sols(D'') \right) \\
\Phi_f(D) =& \{\{D.\texttt{feature}\}\} \diamond \Phi_f(D') \\
\phi_f(D) =& \phi_f(D')
\end{aligned}$$

□

A.2.4 Uniform Sampling

To prove the uniformity of $\mathcal{U}^{FD}$, we first need to introduce lemmas to link the expansion operator with sampling.

Lemma 9. *Let E_1 and E_2 be two sets of configurations on different sets of features, and let $\mathcal{U}^1$ (resp. $\mathcal{U}^2$) be a uniform sampler on E_1 (resp. E_2). Then the sampler defined as*

$$\mathcal{U}(E_1 \diamond E_2) = \mathcal{U}^1(E_1) \cup \mathcal{U}^2(E_2)$$

is a uniform sampler.

Proof. Let $C \in E_1 \diamond E_2$, we want to show that

$$\mathbb{P}\left(\mathcal{U}(E_1 \diamond E_2) = C\right) = \frac{1}{|E_1 \diamond E_2|}$$

Since E_1 and E_2 have different sets of features, a sampled configuration can be uniquely divided into two sub-configurations $C = C_1 \cup C_2$ such that $C_1 \in E_1$ and $C_2 \in E_2$. Then

$$\begin{aligned}
\mathbb{P}\left(\mathcal{U}(E_1 \diamond E_2) = C\right) &= \mathbb{P}\left(\mathcal{U}^1(E_1) \cup \mathcal{U}^2(E_2) = C_1 \cup C_2\right) \\
&= \mathbb{P}\left(\mathcal{U}^1(E_1) = C_1 \wedge \mathcal{U}^2(E_2) = C_2\right) \\
&= \mathbb{P}\left(\mathcal{U}^1(E_1) = C_1\right) \cdot \mathbb{P}\left(\mathcal{U}^2(E_2) = C_2\right) && \text{independency} \\
&= \frac{1}{|E_1|} \cdot \frac{1}{|E_2|} \\
&= \frac{1}{|E_1 \diamond E_2|} && \text{by Lemma 8}
\end{aligned}$$

□

Lemma 10. *Let S be a set of n elements, and c be an element not in S. If $\mathcal{U}$ is a uniform sampler on S, then $\mathcal{U}'$ is defined as*

$$\mathcal{U}'(S \cup \{c\}) = \begin{cases} c & \text{with probability } \frac{1}{n+1} \\ \mathcal{U}(S) & \text{otherwise} \end{cases}$$

Proof. By definition,

$$\mathbb{P}\left(\mathcal{U}'(S \cup \{c\}) = c\right) = \frac{1}{n+1}$$

and

$$\begin{aligned}\forall s \in S, \mathbb{P}\left(\mathcal{U}'(S \cup \{c\}) = s\right) &= \mathbb{P}\left(\mathcal{U}'(S \cup \{c\}) \neq c\right) \cdot \mathbb{P}\left(\mathcal{U}(S) = s\right) \\ &= \frac{n}{n+1} \cdot \frac{1}{n} \\ &= \frac{1}{n+1}\end{aligned}$$

□

Lemma 11. *Let S be a set of n elements, and let $s \in S$. If $\mathcal{U}$ is a uniform sampler on S. To define $\mathcal{U}'$ on $S \backslash \{c\}$ we first sample s' from S, and define $\mathcal{U}'$ as*

$$\mathcal{U}'(S \backslash \{s\}) = \begin{cases} s' & \text{if } s' \neq s \\ \mathcal{U}'(S \backslash \{s\}) & \text{otherwise} \end{cases}$$

Then, $\mathcal{U}$ is a uniform sampler in the set $S \backslash \{s\}$.

Proof. At each step, there is a probability of $\frac{1}{n}$ of sampling s from S, which we do not want. Otherwise there are $\frac{1}{n}$ chances of picking every other element.

$$\begin{aligned}\forall s' \in S \backslash \{s\}, \mathbb{P}\left(\mathcal{U}'(S \backslash \{s\}) = s')\right) &= \frac{1}{n} + \frac{1}{n} \cdot \frac{1}{n} + \frac{1}{n}\left(\frac{1}{n}\right)^2 + \dots \\ &= \frac{1}{n} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{n}\right)^i \\ &= \frac{1}{n} \cdot \frac{1}{1 - \frac{1}{n}} \\ &= \frac{1}{n-1}\end{aligned}$$

□

From these three lemmas we can then prove that the sampler we propose for feature diagrams is uniform.

Proposition 4 (Uniform Sampler on Feature Diagrams)**.** *Given a feature diagram D, the following recursively defined algorithm $\mathcal{U}^{FD}$ is a uniform sampler.*

- *If $D.\texttt{children} = \emptyset$, then $\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\}$*

- *If* $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$,

$$\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\} \cup \bigcup_{D' \in D.\texttt{mand}} \mathcal{U}^{FD}(D') \cup \bigcup_{D' \in D.\texttt{opt}} \begin{cases} \emptyset & \text{with probability } \frac{1}{|Sols(D')|+1} \\ \mathcal{U}^{FD}(D') & \text{otherwise} \end{cases}$$

- *If* $D'.\texttt{xor} \neq \emptyset$, *choose* $D' \in D.\texttt{xor}$ *with probability* $\frac{|Sols(D')|}{|Sols(D)|}$, *then*

$$\mathcal{U}^{FD}(D) = \{D.\texttt{feature}\} \cup \mathcal{U}^{FD}(D')$$

- *If* $D.\texttt{or} \neq \emptyset$, *we define*

$$C = \bigcup_{D' \in D.\texttt{or}} \begin{cases} \emptyset & \text{with probability } \frac{1}{|Sols(D')|+1} \\ \mathcal{U}^{FD}(D') & \text{otherwise} \end{cases}$$

and

$$\mathcal{U}^{FD}(D) = \begin{cases} \{D.\texttt{feature}\} \cup C & \text{if } C \neq \emptyset \\ \mathcal{U}^{FD}(D) & \text{otherwise} \end{cases}$$

Proof. We use Property 14 and the previous lemmas.

- If $D.\texttt{children} = \emptyset$, $Sols(D) = \{\{D.\texttt{feature}\}\}$, so there is only one solution to sample.
- If $D.\texttt{mand} \cup D.\texttt{opt} \neq \emptyset$, then we use Lemma 9, and for the $D.\texttt{opt}$ children we also use Lemma 10
- If $D.\texttt{xor} \neq \emptyset$, we first choose a child D' with probability $\frac{|Sols(D')|}{|Sols(D)|}$, and then uniformly chooses a solution in D'. The probability of choosing any solution is then

$$\begin{aligned} \forall s \in Sols(D), \mathbb{P}\left(\mathcal{U}^{FD} = s\right) &= \mathbb{P}(s \in Sols(D') \wedge \mathcal{U}(D') = s) \\ &= \mathbb{P}(s \in Sols(D')) \cdot \mathbb{P}(\mathcal{U}(D') = s) \\ &= \frac{|Sols(D')|}{|Sols(D)|} \cdot \frac{1}{|Sols(D')|} \\ &= \frac{1}{|Sols(D)|} \end{aligned}$$

- If $D.\texttt{or} \neq \emptyset$, we use Property 14, lemmas 9 and 11.

□

A.3 Diversity

A.3.1 Diversity constraints

We first recall the definitions of the constraints, already presented in Chapter 8.

Definition 38 (Diversity constraints)**.** Let δ be a distance function, $\mathcal{A}$ be an aggregator, and k be an integer. Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ and d be variables, and let $\mathcal{S}$ be a set of k solutions. The $\texttt{single\_diversity}_{\mathcal{A},\delta}$ constraint considers the distance from one set of variables to multiple solutions already found:

$$\texttt{single\_diversity}_{\mathcal{A},\delta}(\mathcal{X}, \mathcal{S}, d) \Leftrightarrow \underset{S \in \mathcal{S}}{\mathcal{A}} \delta(\mathcal{X}, S) \geq d \tag{8.1}$$

For $1 \leq j \leq k$, let $\mathcal{X}^j$ be k sets of n variables, and d be a variable. The $\texttt{multiple\_diversity}_{\mathcal{A},\delta}$ constraint considers the distance between the sets of variables $\mathcal{X}^1, \ldots, \mathcal{X}^k$, i.e. $\delta(\mathcal{X}^i, \mathcal{X}^j)$ is the distance between the i-th and the j-th duplicated solution:

$$\texttt{multiple\_diversity}_{\mathcal{A},\delta}\left(\mathcal{X}^1, \ldots, \mathcal{X}^k, d\right) \Leftrightarrow \underset{1 \leq i < j \leq k}{\mathcal{A}} \delta\left(\mathcal{X}^i, \mathcal{X}^j\right) \geq d \tag{8.2}$$

We recall that for a separable distance δ, we note $\widetilde{\delta}$ the distance on each dimension, such that $\delta(a, b) = \sum_{i=1}^{n} \widetilde{\delta}(a_i, b_i)$.

Definition 40. Let δ be a separable distance function and k be an integer. Let X and d be a variable, $s_1, \ldots, s_k$ be integers, and $X_1, \ldots, X_k$ be variables. We define the diversity constraints on a single dimension:

$$\texttt{single\_diversity\_dim}_{\delta}(X, s_1, \ldots, s_k, d) \Leftrightarrow \sum_{i=1}^{k} \widetilde{\delta}(X, s_i) \geq d \tag{8.5}$$

$$\texttt{multiple\_diversity\_dim}_{\delta}(X_1, \ldots, X_k, d) \Leftrightarrow \sum_{1 \leq i < j \leq k} \widetilde{\delta}(X_i, X_j) \geq d \tag{8.6}$$

We now prove the propositions stated in Chapter 8.

Proposition 6. *All the values are arc consistent on the constraint* $\texttt{single\_diversity}_{\Sigma,\delta}$ *(resp.* $\texttt{multiple\_diversity}_{\Sigma,\delta}$*) iff the network of constraints of the reformulation in equation 8.7 (resp. equation 8.8) is arc consistent.*

Proof. Let $\mathcal{X}$ be a set of variables, d be a variable and $\mathcal{S}$ be a set of solutions. We show the proof for the $\texttt{single\_diversity}_{\Sigma,\delta}(\mathcal{X}, \mathcal{S}, d)$ constraint, it is similar for the

$\texttt{multiple\_diversity}_{\Sigma,\delta}$ constraint. We remark that the variables d_i are intermediate variables, so we do not consider them in the arc consistency: the only interesting variables are those in $\mathcal{X}$ and d. To prove the equivalence, we prove both implications.

AC on $\texttt{single\_diversity}_{\Sigma,\delta}$ $\Rightarrow$ AC on the reformulation We suppose that $\texttt{single\_diversity}_{\Sigma,\delta}$ is arc consistent. Let Z be a variable of the scope of the constraint (remark that Z can be a variable of $\mathcal{X}$, or the variable d) and $z \in \mathcal{D}(Z)$. Since $\texttt{single\_diversity}_{\Sigma,\delta}$ is arc consistent, there exists a tuple τ such that $\tau(Z) = z$, $\forall X \in \mathcal{X}, \tau(X) \in \mathcal{D}(X)$, and $\tau \in rel(\texttt{single\_diversity}_{\Sigma,\delta})$. From τ we build supports for each constraint of the reformulation. Let τ_i containing the variables X_i and d_i such that $\tau_i(X_i) = \tau(X_i)$, and $\tau_i(d_i) = \sum_{j=1}^{k} \widetilde{\delta}(X_i, S_j[i])$. The τ_i supports are supports for the constraints $\texttt{single\_diversity\_dim}_{\delta}(X_i, S_1[i], \dots S_k[i], d_i)$ for all $i \in \{1, \dots, n\}$. Let τ' such that $\tau'(d_i) = \sum_{j=1}^{k} \widetilde{\delta}(X_i, S_j[i])$ and $\tau'(d) = \tau(d)$. τ' is a support for the constraint $\sum_{i=1}^{n} d_i \geq d$. Thus, from a support for a variable of $\texttt{single\_diversity}_{\Sigma,\delta}$, we can build support tuples for all the constraints of the reformulation containing the value z for Z, so the reformulation is arc consistent.

AC on the reformulation $\Rightarrow$ AC on $\texttt{single\_diversity}_{\Sigma,\delta}$ We suppose that all the constraints of the reformulation are arc consistent. Let Z be a variable, and $z \in \mathcal{D}(Z)$ a value. We show how to build a support τ for the $\texttt{single\_diversity}_{\Sigma,\delta}$ constraint such that $\tau(Z) = z$. We need to build the appropriate supports τ_i and τ' from the constraints of the reformulation to generate τ. There are two cases, depending on whether $Z \in \mathcal{X}$ or Z is the variable d.

- If Z is the variable d: we use the arc consistency of the constraint $\sum_{i=1}^{n} d_i \geq d$ to build a support τ' for the value z of Z (i.e. such that $\tau'(Z) = z$). Then we can build supports from all the diversity constraints on each dimension τ_i such that $\tau_i(d_i) = \tau'(d_i)$.
- If $Z \in \mathcal{X}$, then Z is a variable X_j for some j. We build the support τ_j using arc consistency of z on the j-th dimension diversity constraint, such that $\tau_j(Z) = z$. We then use arc consistency on the constraint $\sum_{i=1}^{n} d_i \geq d$ to build a support τ' such that $\tau'(d_j) = \tau_j(d_j)$. Then we build supports for all the remaining diversity constraints τ_i $(i \neq j)$ such that $\tau_i(d_i) = \tau'(d_i)$.

From the supports τ_i and τ', we build the support τ such that $\tau(X_i) = \tau_i(X_i)$ and $\tau(d) = \tau'(d)$. In this support, $\tau(Z) = z$ (by construction of the smaller supports). We

have therefore proved that z is arc consistent with the diversity constraint. □

Theorem 9. *For any $p \geq 0$, arc consistency is NP-hard to propagate on* $\texttt{single\_diversity}_{\min,\delta_{l_p}}$ *and on* $\texttt{multiple\_diversity}_{\min,\delta_{l_p}}$.

Proof. This proof is similar to the one for the Hamming distance presented in [24]. We reduce 3SAT to our problem. Given a Boolean formula $\phi = C_1 \wedge \ldots \wedge C_k$ on the variables $x_1, \ldots, x_n$, our goal is to create a CSP size at most polynomially larger than ϕ (using the diversity constraint) that has a solution iff the Boolean formula has a solution. To do so, we build k solutions S_l such that the clause C_l is satisfied iff the solution of the CSP is distant of S_l.

We create the variables $X_1, \ldots, X_n$ with domain $\{-1, 1\}$. These variables define solutions of the initial Boolean formula ϕ such that $X_i = 1 \Leftrightarrow x_i = true$. For each clause C_l $(1 \leq l \leq k)$ we build a tuple S_l such that $S_l[i] = -1$ if the variable x_i appears as a positive literal in C_l, $S_l[i] = 1$ if x_i appears as a negative literal in C_l, and $S_l[i] = 0$ otherwise. For example, if $C_l = x_{i_1} \vee x_{i_2} \vee \neg x_{i_3}$, we build the tuple S_l where $S_l[i_1] = -1$, $S_l[i_2] = -1$, and $S_l[i_3] = 1$, otherwise 0. We create the constant variable $d = 2 + \left\lfloor (n-3)^{1/p} \right\rfloor$. We create the constraint $\texttt{single\_diversity}_{\min,\delta_{l_p}}(\{X_1, \ldots, X_n\}, \{S_1, \ldots, S_l\}, d)$. This constraint has a solution iff the 3SAT formula has a satisfying assignment:

- If the assignment does not satisfy the model because of clause l, then $\delta_{l_p}((X_i)_i, S_l) = (n-3)^{1/p} < d$, so the instantiation does not satisfy the constraint.
- If the assignment satisfies the model, then for $1 \leq l \leq k$, $\delta_{l_p}((X_i)_i, S_l) \geq (2^p + n - 3)^{1/p} \geq 2 + (n-3)^{1/p} \geq d$ (by Cauchy-Schwarz inequality). The instantiation satisfies the constraint.

We showed that it is NP-hard to say whether the constraint has a satisfying instantiation or not. It is thus NP-hard to ensure that all values of all variables have a support. □

A.3.2 Approximation algorithms

Proposition 10 (1/2-approximation)**.** *When using the* min *aggregator, the greedy and hybrid approaches are 1/2-approximations of the* `MaxDiverseKSet` *problem, i.e. if $\mathcal{S}_{opt}$ is the optimal diverse set of size k, d_{opt} is the minimum pairwise distance of $\mathcal{S}_{opt}$, $\mathcal{S}_g$ is the set returned by the greedy or hybrid approach, and d_g its minimum pairwise distance, then $\frac{1}{2}d_{opt} \leq d_g \leq d_{opt}$*

Proof. We give the proof for the greedy approach. It can be extended to the hybrid approach.

Let $\mathcal{S}_{opt}$ be an optimal solution, and $d_{opt} = \min_{s,s' \in \mathcal{S}_{opt}} \delta(s, s')$ the minimum pairwise distance. Let $\mathcal{S}_g$ be the solution returned by the greedy approach, and d_g the minimum pairwise distance in $\mathcal{S}_g$.

We want to prove that $d_{opt}/2 \leq d_g$. We proceed by contradiction, so we suppose that $d_{opt}/2 > d_g$. We note $\mathcal{S}_g^-$ the set $\mathcal{S}_g$ where the last solution found has been removed (equivalently, $\mathcal{S}_g^-$ is the greedy solution of size $k-1$). We assume without loss of generality that the last solution found has a distance of d_g with another solution (we can just proceed by induction to restrict ourselves to this case). By property of the greedy algorithm, this means that all the solutions to the problem have a distance less than or equal to d_g to the solutions in $\mathcal{S}_g^-$. In particular, it means that for all the solutions $s \in \mathcal{S}_{opt}$, $\exists s_g \in \mathcal{S}_g^-$ such that $\delta(s, s_g) \leq d_g < d_{opt}/2$. There is a unique such solution s_g in $\mathcal{S}_g^-$ which achieves this distance because for $s, s' \in \mathcal{S}_{opt}$, if $\exists s_g \in \mathcal{S}_g^-$ such that $\delta(s, s_g) \leq d_g$ and $\delta(s', s_g) \leq d_g$, then $\delta(s, s') \leq \delta(s, s_g) + \delta(s', s_g) \leq 2 * d_g < d_{opt}$, but $\delta(s, s') \geq d_{opt}$ by definition of d_{opt}. We proved that every solution in $\mathcal{S}_{opt}$ can be uniquely linked to a solution in $\mathcal{S}_g^-$, so $|\mathcal{S}_{opt}| \leq |\mathcal{S}_g^-| = k - 1$, which is a contradiction.

We have proved that there necessarily exists a solution whose distance to other solutions is greater than or equal to $d_{opt}/2$, and solving the MostDistant($\mathcal{S}_g^-$) will necessarily find one. This proves that the greedy approach is 2-optimal. □

Proposition 11. *There exist problems where the minimum distance of the solution returned by the greedy or hybrid approaches is exactly half the minimum distance of an optimal solution.*

Proof. We show a proof using the Hamming or Manhattan distance δ.

For the hybrid approach, we show an example with $k' = k - 1$. We consider the space $\{0, 1\}^{2k'}$. We construct k' solutions $s_0, \ldots, s_{k'-1}$ where

$$\forall i \in \{1, \ldots, k'\}, j \in \{1, \ldots, 2k'\}, s_i[j] = \begin{cases} 1 & \text{if } \lfloor j/2 \rfloor == i \\ 0 & \text{otherwise} \end{cases}.$$

We also construct $2k'$ solutions $s'_1, \ldots, s_{2k'}$ where

$$\forall i \in \{1, \ldots, 2k'\}, j \in \{1, \ldots, 2k'\}, s'_i[j] = \begin{cases} 1 & \text{if } j == i \\ 0 & \text{otherwise} \end{cases}.$$

We note $S = \{s_1, \ldots, s_{k'}\}$ and $S' = \{s'_1, \ldots, s'_{2k'}\}$ and $\mathcal{S} = S \cup S'$. For example, if $k' = 3$,

the set S and S' are as follows:

$$S = \left\{ \begin{array}{l} (1,1,0,0,0,0), \\ (0,0,1,1,0,0), \\ (0,0,0,0,1,1) \end{array} \right\} \text{ and } S' = \left\{ \begin{array}{l} (1,0,0,0,0,0), \\ (0,1,0,0,0,0), \\ (0,0,1,0,0,0), \\ (0,0,0,1,0,0), \\ (0,0,0,0,1,0), \\ (0,0,0,0,0,1) \end{array} \right\}.$$

We have some properties on the distances:

- $\forall s, s' \in S$ such that $s \neq s', \delta(s, s') = 4$,
- $\forall s, s' \in S'$ such that $s \neq s', \delta(s, s') = 2$,.
- $\forall s' \in S', \forall s \in S, \delta(s', s) \in \{1, 3\}$,

When the hybrid approach is applied on the set $\mathcal{S}$, it first solves exactly for k' solutions, returning the set S (of minimum distance 4). It will then greedily choose a solution s', necessarily in S'. The resulting set has a minimum distance of 1, because there is a solution s in S such that $\delta(s, s') = 1$.

However, an optimal solution set is to pick k solutions from S', giving a minimum distance of 2.

We were able to build a solution set so that the minimum distance is half the minimum distance of an optimal solution.

Remark. *It is possible to extend this proof to deal with the general case of any combination of k' and k in the hybrid or greedy approach.*

□

Appendix B

Supplementary Material for TableSampling

This appendix presents in Table B.1 the running times on all instances of the MiniZinc benchmark.

Table B.1 – Running times of sampling approaches on the MiniZinc benchmark. A ⊥ indicates a timeout, a ∗indicates an error (`java.lang.OutOfMemoryError: Java heap space`). Dicho is the dichotomic variation of the base algorithm.

Year	Problem	Instance	Random-Search	TableSampling				Lin-ModEq
				NoPropag		Propag		
				Base	Dicho	Base	Dicho	
2016	elitserien	handball3	⊥	72.6s	137s	75.8s	143s	⊥
		handball7	⊥	52.7s	166s	77.8s	169s	⊥
		handball5	⊥	182s	410s	305s	357s	⊥
		handball20	⊥	36.9s	115s	37.1s	64.2s	⊥
	mrcpsp	j30_15_5	22.6s	130s	⊥	39.2s	35.2s	⊥
		j30_17_10	1.14s	1.96s	5.82s	444ms	718ms	⊥
	depot-placement	rat99_6	585s	299s	587s	390s	⊥	⊥
		rat99_5	11.7s	22.9s	64.5s	22.7s	50.1s	135s
		st70_5	67.9s	130s	⊥	147s	293s	⊥
		ulysses22_5	56.1s	221s	291s	150s	113s	⊥
	java-auto-gen	plusexample_6	⊥	137s	258s	123s	161s	535s
		binpack_11	⊥	21.5s	13.8s	8.57s	15.9s	⊥
	filters	fir_1_3	53.3ms	238ms	370ms	621ms	248ms	⊥
		dct_1_3	1.86s	⊥	⊥	⊥	⊥	⊥
		fir_1_4	55.1ms	262ms	441ms	681ms	259ms	⊥
		ar_1_3	4.23s	13.4s	33.7s	13.9s	17.2s	⊥
		ewf_1_2	116s	471ms	936ms	259ms	360ms	⊥

	zephyrus	14___6___8___3	3.05s	479s	⊥	247s	367s	⊥
		12___6___8___3	2.96s	⊥	⊥	⊥	⊥	⊥
		12___8___6___3	2.97s	318s	⊥	629s	⊥	⊥
		14___8___6___3	1.92s	220s	366s	121s	173s	⊥
	rcpsp-wet	j30_27_5-wet	⊥	502s	⊥	377s	⊥	⊥
		j30_44_8-wet	67.6s	160s	292s	201s	464s	156s
	carpet-cutting	mzn_rnd_test.11	⊥	⊥	334s	⊥	165s	⊥
2017	routing-flexible	routing_GCM_0022	122s	*	*	⊥	⊥	*
	groupsplitter	u6g3pref0	⊥	*	⊥	⊥	*	⊥
		u12g1pref0	33.6s	⊥	⊥	137s	296s	161s
		u5g1pref0	3.11s	18.8s	73.0s	8.93s	31.1s	1.36s
		u12g1pref1	20.4s	210s	387s	103s	118s	146s
	community-detection	Sampson.s10.k3	⊥	⊥	⊥	⊥	⊥	⊥
	steelmillslab	bench_16_10	⊥	⊥	⊥	⊥	⊥	⊥
		bench_14_1	⊥	⊥	⊥	⊥	⊥	⊥
	tc-graph-color	k5_05	⊥	60.5s	162s	70.0s	146s	101s
		k10_31	⊥	296s	⊥	287s	⊥	373s
	opd	small_bibd_08_28_14	⊥	⊥	⊥	⊥	⊥	⊥
		small_bibd_06_50_25	⊥	⊥	⊥	⊥	⊥	⊥
2018	elitserien	handball18	⊥	45.6s	159s	48.2s	100s	⊥
		handball1	⊥	154s	373s	158s	165s	⊥
		handball15	⊥	101s	352s	117s	163s	⊥
		handball6	⊥	50.1s	108s	46.5s	97.9s	⊥
		handball16	⊥	215s	694s	290s	432s	⊥
	soccer-computationa	xIGData_22_12_22_5	⊥	⊥	⊥	⊥	⊥	⊥
	test-scheduling	t100m10r3-2	⊥	*	*	⊥	*	⊥
		t30m10r10-5	⊥	*	*	⊥	⊥	⊥
	neighbours	neighbours1	22.1s	82.6s	113s	112s	113s	87.6s
	steiner-tree	es10fst10.stp	8.92s	37.2s	57.5s	37.3s	62.3s	23.9s
		es10fst03.stp	4.57s	8.56s	13.9s	6.71s	14.1s	5.71s
	racp	j30_26_2_1.0	⊥	269s	⊥	189s	⊥	⊥
	rotating-workforce	Example103	⊥	⊥	⊥	⊥	⊥	⊥
		Example1479	⊥	⊥	⊥	⊥	⊥	⊥
	concert-hall-cap	concert-cap.mznc201	72.8s	71.1s	157s	82.2s	65.9s	510s

		concert-cap.mznc201	3.52s	12.8s	20.5s	7.85s	10.2s	11.1s
	oocsp_racks	oocsp_racks_100_r1_	⊥	⊥	⊥	⊥	⊥	⊥
		oocsp_racks_050_r1	⊥	⊥	⊥	⊥	⊥	⊥
2019	stack-cuttingstock	d3	701ms	3.29s	8.58s	5.16s	7.25s	11.9s
	groupsplitter	u6g1pref1	4.21s	73.2s	70.7s	20.4s	37.8s	⊥
		u9g1pref1	17.4s	145s	257s	60.8s	94.1s	80.2s
	stochastic-vrp	vrp-s4-v2-c3_svrp-v	4.20s	378ms	656ms	308ms	445ms	123ms
	steelmillslab	bench_19_6	⊥	⊥	⊥	⊥	⊥	⊥
		bench_20_8	⊥	⊥	⊥	⊥	⊥	⊥
	median-string	p2_10_8-0	⊥	97.2s	160s	87.6s	136s	291s
	zephyrus	12___8___6___3	3.02s	308s	⊥	632s	⊥	⊥
		14___6___6___3	761ms	5.78s	9.35s	3.26s	3.66s	⊥
		12___6___6___3	862ms	4.48s	8.01s	2.40s	2.57s	⊥
2020	is	A3PZaPjnUz	⊥	*	*	3.56s	6.64s	⊥
		v1HjuSBQMb	127s	*	*	5.44s	8.97s	⊥
	soccer-computationa	xIGData_22_12_22_5	⊥	⊥	⊥	⊥	⊥	⊥
	collaborative-const	46	⊥	⊥	⊥	⊥	⊥	⊥
	bnn-planner	cellda_y_10s	⊥	121s	198s	251s	268s	⊥
	p1f-pjs	10	⊥	123s	89.0s	124s	101s	⊥
	skill-allocation	skill_allocation_mz	⊥	⊥	⊥	⊥	⊥	⊥
	pentominoes	05	⊥	⊥	⊥	485s	⊥	⊥
		04	⊥	⊥	⊥	⊥	⊥	⊥
		02	7.60s	14.5s	41.8s	63.7s	27.3s	⊥
		06	⊥	⊥	⊥	⊥	⊥	⊥
		07	⊥	⊥	⊥	⊥	⊥	⊥
	racp	j30_26_2_1.0	⊥	271s	⊥	185s	⊥	⊥
	minimal-decision-se	breast-cancer_train	⊥	74.5s	120s	97.5s	122s	⊥
2021	peacable_queens	8	47.4s	168s	⊥	209s	498s	609s
	opt-cryptoanalysis	r2	151ms	42.4ms	40.8ms	126ms	65.8ms	196ms
		r4	31.8s	2.34s	9.75s	3.04s	2.56s	⊥
		r3	390ms	93.5ms	93.4ms	107ms	136ms	13.3s
		r1	88.1ms	46.5ms	29.6ms	100ms	83.5ms	64.7ms

Appendix C

Logic Games Solver

This chapter presents the models implemented in the application Logic Games Solver. I developed this application for the CP app competition that took place during the conference CP 2022. The application won a honourable mention (i.e. the first place). The app is available in the Android Play Store [a].

a. https://play.google.com/store/apps/details?id=com.mvavrill.logicGamesSolver

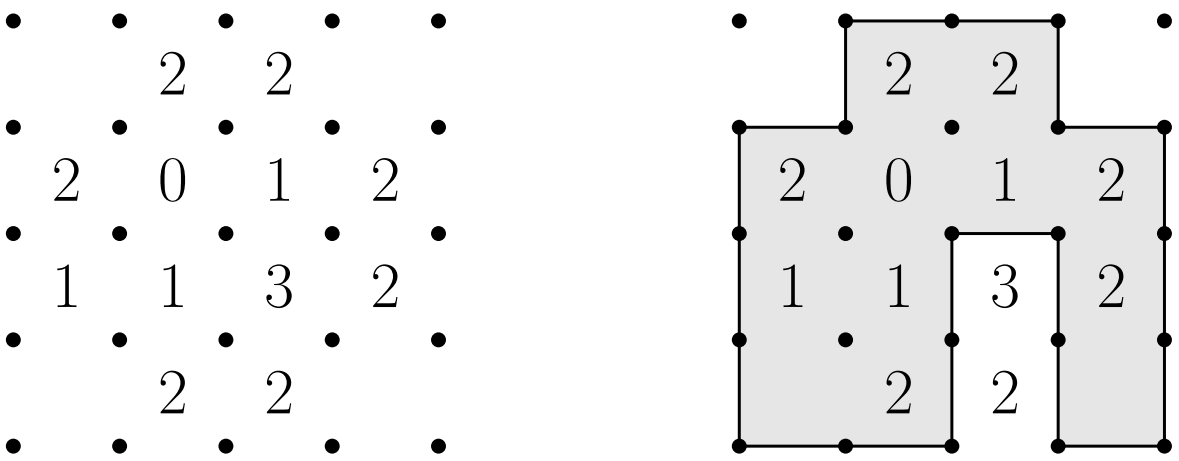
C.1 Introduction

Logic games are single player games where, from an initial grid, a *single* solution can be created satisfying the rules of the game. Constraint Programming is a great tool to solve logic games because the rules are often stated as constraints. I present here four logic games and their CP model.

C.2 Slitherlink

C.2.1 Rules

Slitherlink is a game where, from a grid of dots and the goal is to connect the dots (vertically and horizontally) to make a *single* loop. The digits indicate the number of edges around the clue. The loop does *not* have to touch every dot. An example of input and solution is given in Figure C.1. Remark that a single loop defines one *inside* and one *outside* (due to Jordan curve theorem). For example, in Figure C.1b, the inside is coloured in grey.



(a) Input grid (b) Solution

Figure C.1 – Example of *Slitherlink* input and its solution.

C.2.2 CP Model

We first present a working model, and then present improvements by adding expert knowledge.

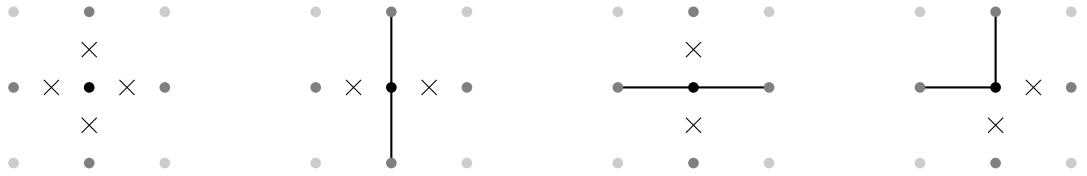
A First Model

The game's solution loop can be seen as a cycle in a graph. We use graph variables [16] where each dot of the game is a vertex, and the link between two dots is a possible edge. Using the graph variable makes the model very simple:

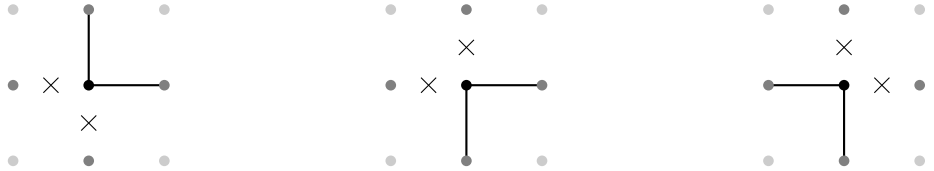
- The graph should be a cycle.
- For every clue, the number of edges present around the clue is fixed.

Specifically, we create a graph variable (i.e. a graph whose edges are boolean variables) where each vertex of the graph is a node of the game. For each node *node*, we note *node.top* (resp. $node.bottom, node.left, node.right$) the edge variable on top (resp. bottom, left, right) of the node. Then, the graph induced by the node and the edges should be a cycle. To do so, the `cycle` constraint can be used. It ensures that the graph contains only a single cycle. It allows to filter edges that would make a sub-cycle.

To ease the notations, we note *cell* a structure representing a cell of the game (between four nodes). In this structure, the attribute *cell.clue* contains the clue (or -1 if the cell is empty). It also contains the attribute *cell.ul* (resp. $cell.ur, cell.br, cell.bl$) for the the upper left (resp. upper right, bottom right, and bottom left) node of the cell. Finally, the structure contains the attribute *cell.top* (resp. $cell.bottom, cell.left, cell.right$) to store the



(a) Pattern 0 (b) Pattern 1 (c) Pattern 2 (d) Pattern 3



(e) Pattern 4 (f) Pattern 5 (g) Pattern 6

Figure C.2 – The seven patterns for the loop passing (or not) through a node. A cross indicates the absence of an edge.

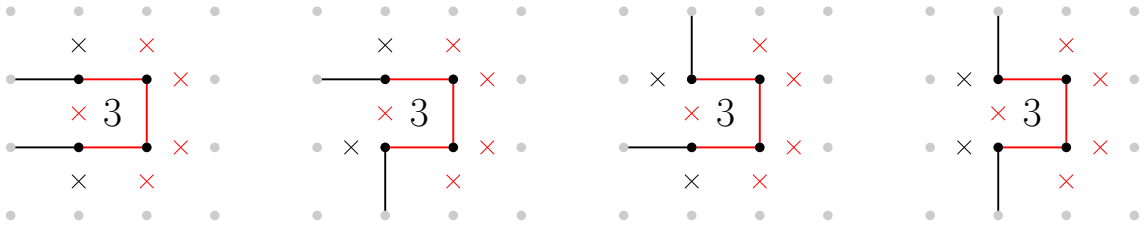
edge at the top (resp. bottom, left and right) of the cell. The constraint for the clues is then: $\forall cell$, if $cell.clue \neq -1$, then $cell.top + cell.bottom + cell.left + cell.right = cell.clue$.

This simple model is sufficient to find the solution. However, without guidance, it is rare to be able to draw (or remove) a link (except for the 0 clues) without decisions and backtracks. To improve the model (make the solving faster), it is possible to add expert knowledge.

C.2.3 Expert Knowledge

I present here a way to add redundant constraints to the model. These redundant constraints do not further constrain the solution (as it is already found by the base model), but help the solver propagate more information (instead of making decisions and backtracks). To add this expert knowledge, we create a variable for each node, noted *node.var*. There are only a few ways the loop can pass through a node, we enumerate these ways and associate them with an integer as in Figure C.2. Graphically, the absence of an edge is drawn using a cross.

The variables *node.var* take values in the set $\{0, \ldots, 6\}$ and model the pattern of the node. These patterns need to be associated to the edge variables. This is done using a `table` constraint, which contains a tuple for each pattern. For each node *node*, the



(a) $(2, 6, 3, 2)$ (b) $(2, 6, 3, 5)$ (c) $(4, 6, 3, 2)$ (d) $(4, 6, 3, 5)$

Figure C.3 – Possible patterns (in black) for clue three such that $cell.left = 0$ (red edges and crosses are fixed). The sub-captions give the values associated to $(cell.ul, cell.ur, cell.br, cell.bl)$.

following constraint is added

$$\texttt{table}(node.var, node.left, node.top, node.right, node.bottom, \left\{ \begin{array}{l} (0,0,0,0,0), \\ (1,0,1,0,1), \\ (2,1,0,1,0), \\ (3,1,1,0,0), \\ (4,0,1,1,0), \\ (5,0,0,1,1), \\ (6,1,0,0,1) \end{array} \right\})$$

This `table` constraint links the node pattern to its adjacent edges. For the nodes in the border of the game, either special tables with fewer variables can be created, or *fake* edges that take the values 0 can be used (to replace of the non-existent edge outside the game).

These patterns can then be linked to the given clues. For example, Figure C.3 enumerates the possibilities for a $cell.clue = 3$ and $cell.left = 0$. There are 4 ways to arrange the loop. The possibilities should also be enumerated when the top, bottom and right edges are absent. This makes 16 possible patterns for the tuple of variables $(cell.ul, cell.ur, cell.br, cell.bl)$. These tuples can be put into a `table` constraint. This also makes the constraint $cell.top + cell.bottom + cell.left + cell.right = cell.clue$ redundant, as all the possibilities are enumerated.

For all possible clue values (either 0, 1, 2, or 3), all the possibilities should be enumerated and a table constraint added. The tables contain:

- 16 tuples for a clue 0,
- 64 tuples for a clue 1,
- 64 tuples for a clue 2,

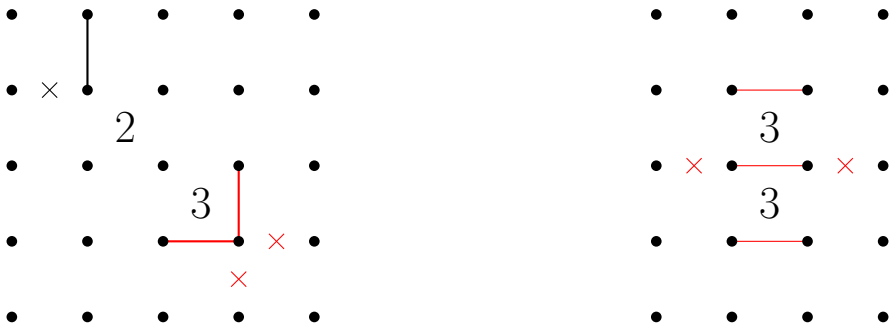
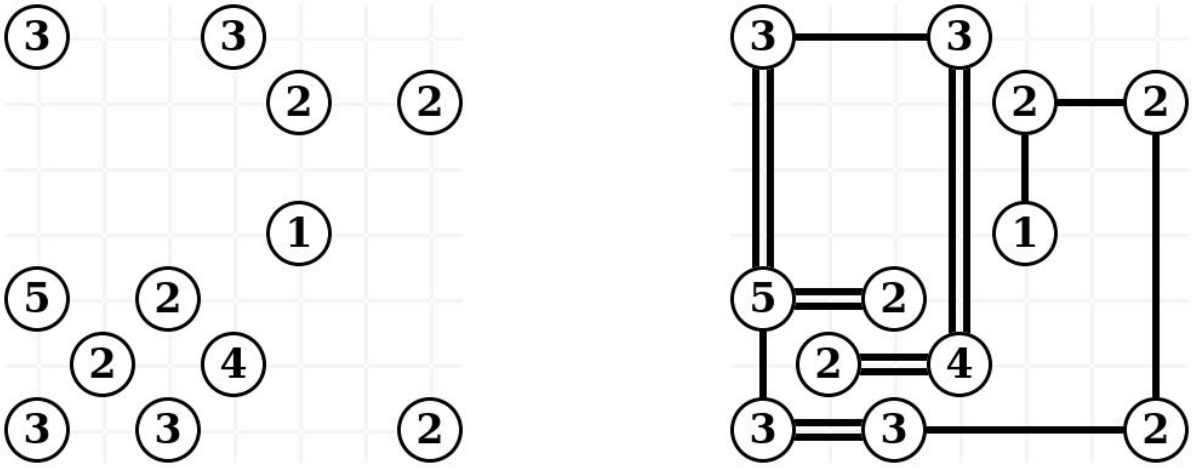


Figure C.4 – Examples of propagation using expert knowledge. The edges (and crosses) in black are the already found information, and the edges (and crosses) are the information that can be extracted (i.e. propagated by the model).



(a) Input grid

(b) Solution

Figure C.5 – Example of *Bridges* input and its solution.

- 16 tuples for a clue 3.

Once all these constraints have been added, the solver will be able to propagate more information. For example, Figure C.4 shows two examples of propagation performed by the model with the expert knowledge.

C.3 Bridges

C.3.1 Rules

Bridges (also called *Hashi*) is a game about an archipelago. A group of islands (the input numbers) should be connected by bridges (only vertically and horizontally), so that in the end, all the islands are connected (by a path of bridges) and that there are no cycles. The numbers in the islands are the number of bridges connected to the island. From one island to another, there can be either zero, one or two bridges. An example of solution is given in Figure C.5.

C.3.2 CP Model

Bridges is also a game with an underlying graph structure. This time, the graph has a tree structure (i.e. connected without cycles). There is one vertex for each island, and one edge variable for each possible bridge. Given two islands A and B, the edge variable $E_{A,B}$ is equal to one if there is at least one bridge between A and B. On this graph variable, a `tree` constraint can be posted (ensuring that the graph is a tree, i.e. connected and has no cycle). Then, there is also an integer variable to know how many bridges there are between the two islands $X_{A,B} \in \{0, 1, 2\}$. This integer variable is channelled to the edge variable by the constraint $E_{A,B} = 0 \Leftrightarrow X_{A,B} = 0$. This allows to link the graph variable (the edge variables) to the ones counting the number of bridges on each island. The sum of the variables linked to an island should be equal to the number of bridges on that island.

C.4 Kakuro

C.4.1 Rules

In *Kakuro*, the goal is to enter digits from 1 to 9 into the cells. The numbers given in the grid correspond to the sum of the digits in the corresponding row (column). In addition, the digits from the same row or column (i.e. corresponding to the same clue) should all be different.

C.4.2 CP Model

The model contains one variable per empty cell, taking value in the set $\{1, \ldots, 9\}$. For each clue H, the sum of the variables in the row (or column) of that clue should be equal to H, and all these variables should be different. Let $\mathcal{X}$ be the variables associated with the clue. Then the model contains the constraints

$$\sum_{X \in \mathcal{X}} X = H$$
$$\texttt{alldifferent}(\mathcal{X})\,.$$

However, performing arc consistency on these two constraints does not filter out all the impossible values. For example, if three cells must sum to 24 and be different, then the only possible set of values is $\{7, 8, 9\}$. However, as the `sum` and `alldifferent` constraints are separated, the value 6 is kept because the `sum` constraint cannot filter it (because $6 + 9 + 9 = 24$).

To improve the propagation, as suggested in [189], it is possible to merge the two constraints into one `alldifferent_sum` constraint. In the Kakuro, the sums can be at most be 45 (using all values from 1 to 9). It is very easy to enumerate all the possible values for the variables associated with the clues. A table constraint can then be created to represent and propagate the `alldifferent_sum` constraint.

C.5 Sudoku

C.5.1 Rules

The goal is to fill the cells with digits from 1 to 9 so that each row, column and large 3×3 cells contain different digits.

C.5.2 CP Model

The CP model contains one variable $X_{i,j}$ per cell (i, j) that takes values in the set $\{1, \ldots, 9\}$. The model contains 27 `alldifferent` constraints (9 for the rows and columns, and 9 for the big cells).

Annexe D

Résumé Long

Introduction

La répétition est souvent ennuyeuse. Lire un livre pour la deuxième fois n'est pas la même chose, on connaît déjà l'intrigue, la fin et tout ce qui se passe. Dans un parc d'attractions, la deuxième fois que vous montez sur les montagnes russes n'est pas la même. Après chaque tour, la quantité d'émotions diminue. Heureusement, il y a souvent d'autres montagnes russes dans le même parc, et il y a beaucoup de nouveaux livres à lire.

Lorsqu'il · elle crée quelque chose (une nouvelle montagne russe, un livre, de la musique), le · a créateur · ice essaie de trouver de nouvelles façons d'assembler des éléments qui n'ont jamais été vus auparavant. Il · elle essaie de créer quelque chose de nouveau à partir de ce qui existe déjà. Ces dernières années, de nombreux outils d'Intelligence Artificielle (IA) ont été développés et améliorés pour aider à la création dans de nombreux domaines. Par exemple, l'un de mes créateurs de musique électronique préférés, DJ S3RL, a créé une chanson en utilisant uniquement l'IA. Il a réalisé une vidéo sur l'ensemble du processus [1] où il entraîne OpenAI Jukebox [169] en lui donnant tous ses morceaux. OpenAI Jukebox génère alors une nouvelle musique. Cette musique doit être éditée manuellement par DJ S3RL, car elle contient de bonnes mélodies mais manque de structure musicale (BPM fixe, répétition d'un refrain, paroles incompréhensibles). Il est intéressant de noter que DJ S3RL n'est pas un informaticien. Cela signifie que tous les outils qu'il utilise peuvent être utilisés par des non-experts. Tous ces outils d'IA donnent des résultats très divers, de sorte qu'ils peuvent être utilisés par un · e créateur · ice de contenu pour obtenir de nouvelles idées qui peuvent ensuite être améliorées à la main.

1. https://www.youtube.com/watch?v=JChbUcjZUBM

Programmation Par Contraintes

La *Programmation Par Contraintes* (PPC) est une technique de programmation déclarative de l'IA. En tant que technique de programmation déclarative, les propriétés des solutions doivent être définies, mais l'algorithme de recherche est déjà implémenté. En programmation par contraintes, l'utilisateur · ice déclare des *contraintes* qui doivent être satisfaites par un ensemble de *variables*, et un *solveur* de contraintes (solveur PPC) trouve une solution (c'est-à-dire une valeur pour les variables) qui satisfait les contraintes. La programmation par contraintes est très générique, avec de multiples contraintes permettant à l'utilisateur · ice d'énoncer des propriétés de haut niveau sur les solutions. Elle peut être considérée comme une boîte noire : l'utilisateur · ice introduit les contraintes et les algorithmes trouvent une solution, mais le processus de recherche peut également être modifié de multiples façons. Dans ce sens, on peut parler de boîte grise, c'est-à-dire que l'algorithme principal ne peut pas être modifié, mais qu'une API étendue est fournie pour ajuster le comportement.

Nous distinguons trois types de personnes travaillant avec des solveurs de PPC. D'un côté, les utilisateur · ices ont des problèmes réels à résoudre. Dans la programmation par contraintes, nous pensons en termes de solutions : que veulent obtenir les utilisateur · ices de l'algorithme et comment une *solution* peut-elle être décrite. Une fois que les utilisateur · ices ont décrit leur problème (il peut s'agir d'un processus itératif), un · e modélisateur · ice traduit ce problème en un problème de satisfaction de contraintes, c'est-à-dire dans le langage de résolution de la programmation par contraintes. Ce · tte modélisateur · ice doit connaître les fonctions (et les contraintes) fournies par le solveur (soit par le biais d'une API, soit en utilisant des langages de haut niveau tels que MiniZinc [39] ou XCSP^3 [11]). Au cours de cette étape, des choix de modélisation peuvent être faits et des stratégies de recherche peuvent être définies pour ajouter la connaissance du domaine dans le solveur de PPC. À l'autre extrémité de l'application de PPC, le · a développeur · euse du solveur de PPC implémente les outils nécessaires au modélisateur · ice pour trouver les solutions. Le · a développeur · euse doit fournir une API facile à utiliser, mais aussi implémenter tous les algorithmes efficaces de propagation de contraintes dans le solveur.

Dans cette thèse, nous oscillons entre le · a modélisateur · ice et le · a développeur · ice, tout en tenant compte des besoins des utilisateur · ices finaux. Nous voulons définir la diversité d'une manière qui soit facile à utiliser pour un modélisateur · ice, et l'implémenter dans le solveur. Par exemple, dans le chapitre 5, nous fournissons un moyen de générer des solutions de manière aléatoire, et nous l'avons implémenté dans le solveur.

Aléatoire

L'aléatoire est un terme utilisé pour décrire un comportement qui ne peut pas être prédit. Cela commence par un simple jeu de pile ou face. Une pièce équilibrée tombera sur pile une fois sur deux en moyenne. Toutefois, avec de l'entraînement, il est possible d'augmenter les chances d'obtenir un côté choisi en jouant toujours à pile ou face de la même manière. Le lancer d'un dé fonctionne de la même manière. Si les conditions initiales sont les mêmes, le résultat du dé sera toujours le même. Cependant, une petite modification des conditions initiales (un angle de la table, la présence d'un courant d'air) changera complètement le résultat. Ce comportement est chaotique. Il est extrêmement difficile d'étudier avec précision un système chaotique avec une approche déterministe : c'est là qu'intervient l'aléatoire. Au lieu d'essayer d'analyser parfaitement le comportement, il est possible d'analyser le résultat moyen. La célèbre citation d'Albert Einstein à propos de la mécanique quantique est un autre exemple de l'utilisation du hasard comme outil de modélisation : "Dieu ne joue pas aux dés avec l'univers". Le comportement aléatoire des particules quantiques est une approximation de modélisation, car nous ne connaissons pas le comportement sous-jacent exact de ces particules. Dans cette thèse, nous utilisons le hasard pour modifier le comportement d'algorithmes habituellement déterministes. L'utilisation du hasard dans les algorithmes d'optimisation n'est pas nouvelle. Par exemple, le recuit simulé, les algorithmes génétiques et la recherche arborescente de Monte-Carlo (MCTS) utilisent tous le hasard et ne fonctionneraient pas sans lui. Par exemple, dans le MCTS, le futur gagnant d'une position de jeu donnée ne peut être évalué avec précision en raison de l'explosion combinatoire des états d'un jeu. Une partie aléatoire est jouée et, dans de bonnes conditions, un nombre suffisant de ces parties aléatoires permet d'obtenir une bonne évaluation de la position de jeu.

Contributions

Cette thèse porte sur la diversité des solutions dans les solveurs de PPC en utilisant des approches probabilistes. L'algorithme de *backtrack-search* des solveurs de PPC est un cadre puissant mais rigide pour trouver des solutions. Nous proposons des moyens de modifier le comportement du solveur en utilisant l'aléatoire pour générer des solutions diverses. Nous analysons également en détail le comportement de nos algorithmes (et des algorithmes état-de-l'art) afin de comprendre leurs propriétés.

Dans un premier temps nous étudions les échantillonneurs de problèmes de contraintes,

et nous proposons un nouvel échantillonneur, TableSampling, dédié à la PPC. Ensuite nous utilisons les stratégies de recherche comme moyen d'ajouter de l'aléatoire pour trouver des solutions diverses. Enfin, nous étudions en détail les notions de diversité dans les solveurs de contraintes, notamment dans un formalisme multi-objectif.

D.1 Échantillonneurs

Dans un premier temps de cette thèse nous nous concentrons sur les échantillonneurs état-de-l'art. L'échantillonnage consiste à tirer au hasard des solutions d'une famille de solutions. Il s'agit d'une approche probabiliste puissante pour estimer des quantités. Par exemple, une approche visant à estimer le nombre de poissons dans un lac fermé peut être réalisée en deux étapes [171] : tout d'abord, N poissons sont capturés, étiquetés et relâchés. Quelques jours plus tard (pour permettre aux poissons de se mélanger), N poissons sont à nouveau capturés et le nombre de poissons marqués (disons n) est compté. Ainsi, on estime qu'il y a une proportion de n/N poissons marqués dans le lac, alors le nombre total estimé de poissons dans le lac est de N^2/n. Cette méthode d'échantillonnage évite d'assécher le lac et de tuer tous les poissons pour les compter.

Dans les problèmes combinatoires, l'espace des solutions est souvent trop grand pour être énuméré. En théorie, les approches d'échantillonnage peuvent être utilisées pour estimer le nombre de solutions. L'échantillonnage permet également de diversifier les solutions obtenues, par exemple lorsque la diversité ne peut pas être définie formellement ou lorsqu'elle est trop coûteuse à calculer.

Il est très difficile d'échantillonner des solutions parfaitement uniformément, où toutes les solutions ont la même probabilité d'être échantillonnées. L'échantillonnage pondéré est encore plus difficile car il permet aux utilisateurs de définir leur propre distribution de solutions. Dans les problèmes contraints, très peu d'échantillonneurs parviennent à l'uniformité. En revanche, un échantillonnage efficace, même s'il n'est pas exactement uniforme, peut être utilisé comme approximation pour générer plusieurs solutions. Cela conduit à de multiples échantillonneurs.

Nous présentons dans le chapitre 4 de nombreux échantillonneurs de problèmes de contraintes. Nous les avons triés par approche utilisée (contraintes de hachage, comptage, compilation, PPC) et nous les présentons dans les grandes lignes, avec du pseudo-code. Cela permet d'avoir une vue globale des algorithmes d'échantillonnage ainsi que de leurs propriétés.

D.2 TableSampling

En utilisant la satisfaction des contraintes comme technique de base, les solveurs de contraintes ont été enrichis de diverses propriétés supplémentaires, telles que l'optimisation (même avec des objectifs multiples [152]), les préférences de l'utilisateur [51], les solutions diverses [24], les solutions robustes [23], etc. Cependant, il existe très peu de travaux sur la randomisation des solutions dans les solveurs PPC.

Il y a peu d'échantillonneurs de PPC, et ces échantillonneurs ne sont pas conçus comme des améliorations des solveurs PPC, mais plutôt comme des algorithmes distincts : MBE-s [72] et SampleSearch [76] transforment les contraintes en un réseau Bayesien, et MDD-s [85] transforme les contraintes en un MDD (diagramme de décision multi-valué). Ces trois échantillonneurs ne bénéficient pas des améliorations apportées aux solveurs de PPC (comme un meilleur temps d'exécution ou de nouvelles contraintes). Dans ce chapitre, nous proposons une méthode pour échantillonner des solutions à un problème de contraintes, sans modifier son modèle, et en utilisant un solveur PPC comme boîte noire. Ce travail est motivé par de nombreuses situations où l'utilisateur d'un solveur de contraintes a besoin de solutions aléatoires : pour faciliter le retour d'information et la prise de décision de l'utilisateur (en fournissant une variété de solutions, représentatives de l'espace des solutions), pour assurer l'équité (pour éviter les modèles dans les solutions consécutives, par exemple dans les problèmes de planification), ou pour fournir une couverture de solution (par exemple dans les problèmes de génération de tests).

Actuellement, un moyen simple d'échantillonner aléatoirement des solutions avec un solveur PPC consiste à utiliser RandomSearch, c'est-à-dire à sélectionner aléatoirement une variable et une valeur en tant que stratégie d'énumération. Toutefois, cette stratégie ne renvoie pas les solutions uniformément (uniformes au sein de l'ensemble de solutions). Un autre inconvénient majeur de cette technique est que RandomSearch remplace la stratégie qui peut avoir été choisie ou construite pour le problème, ce qui est susceptible d'augmenter le temps de résolution.

Notre approche s'inspire de Unigen [82], un algorithme d'échantillonnage approximativement uniforme pour SAT, et nous l'adaptons au cadre de la PPC. L'idée est de diviser l'espace de recherche en ajoutant des contraintes de hachage aléatoires, jusqu'à ce qu'il ne reste qu'un petit nombre tractable de solutions. Il n'est pas nécessaire de remplacer la stratégie et l'échantillonnage peut être effectué parmi les solutions restantes.

La famille de contraintes de hachage aléatoire choisie a un impact important sur le

temps d'exécution. Pour garder un temps de calcul raisonnable, nous choisissons de générer aléatoirement des contraintes de table [15], qui sont implémentées dans tous les solveurs de contraintes. Nous nous appuyons sur leur représentation en extension des tuples valides pour générer, à faible coût, une distribution uniforme multivariée.

Nous avons implémenté notre proposition dans le solveur `choco-solver` [46] et nous la comparons à RandomSearch sur un large benchmark, construit à partir de la compétition annuelle MiniZinc. Nous montrons que notre approche utilisant les contraintes du tableau améliore, en pratique, la qualité de l'aléatoire par rapport à RandomSearch, tout en échantillonnant plus de problèmes.

Nous appliquons également notre algorithme avec des égalités modulaires linéaires [41], qui sont des contraintes de hachage avec des propriétés théoriques plus fortes en termes d'aléa, mais plus difficiles à propager. Sur notre ensemble de référence, l'utilisation d'égalités modulaires linéaires donne une meilleure qualité d'aléatoire par rapport aux contraintes de table, car elle fournit un échantillonnage uniforme. L'inconvénient est un temps d'exécution plus long.

D.3 Fouille de Données

Récemment, plusieurs problèmes de fouille de données ont été exprimés en programmation par contraintes, ce qui permet aux utilisateurs de définir des requêtes complexes à l'aide de langages de haut niveau [97, 104, 108, 110, 115]. Les solveurs PPC sont modulaires, de sorte que les requêtes peuvent être affinées sans modifier le processus de résolution, contrairement aux algorithmes dédiés à la fouille de données. Des contraintes supplémentaires peuvent facilement être ajoutées pour répondre aux besoins d'un · e utilisateur · ice [113]. Par exemple, le *prix de transaction total* (une limite sur une somme pondérée sur le motif) comme vu dans [96] est gérée de manière native par les solveurs PPC. Plus récemment, Hien et al. [109] ont proposé une contrainte globale pour extraire des motifs intéressants, en veillant à ce que les résultats soient diversifiés par rapport à l'indice de Jaccard, une métrique classique dans l'extraction de motifs. Les auteurs ont dû assouplir le problème pour tenir compte de la non-monotonicité de l'indice de Jaccard, ce qui limite l'efficacité de la contrainte.

Cependant, les bases de données sont souvent très grosses et le nombre de motifs trouvés par les solveurs peut être beaucoup trop important pour être utile. Les experts humains ou les algorithmes de décision ont besoin de petits ensembles de motifs pour

travailler. L'une des contraintes les plus classiques imposées aux motifs est la fréquence. Le problème de l'extraction de motifs fréquents a été introduit en [96] pour la tâche d'extraction de règles d'association. Il permet de trouver de nombreuses relations intéressantes entre les données. Les motifs fréquents présentés à un utilisateur doivent également être *diverses* pour éviter la répétition des informations, qui fait perdre du temps à l'expert ou égare les algorithmes. Une approche classique consiste à exploiter d'abord un vaste ensemble de motifs, puis à sélectionner un bon sous-ensemble. Cependant, la fameuse "explosion de motifs" conduit à des résultats très volumineux qui sont difficiles à traiter a posteriori, en particulier sur des bases de données denses ou de grande taille.

Dans cette thèse, nous utilisons des stratégies de recherche, qui sont classiquement conçues pour améliorer l'efficacité des solveurs, comme moyen d'imposer la diversité dans l'exploration de motifs fréquents. Nous proposons une nouvelle stratégie, OrientedSearch, et une fonction de notation associée, pour orienter la recherche vers des espaces de solution diversifiés. Nous mesurons la diversité à l'aide de l'indice de Jaccard, mais notre approche peut utiliser n'importe quelle mesure de diversité (monotone ou non). Nous avons expérimenté notre approche sur des bases de données denses et peu denses. Les expériences montrent que l'utilisation de stratégies de recherche aléatoire (RandomSearch ou l'approche proposée OrientedSearch) améliore de manière significative la diversité des motifs retournés par rapport aux autres approches état-de-l'art. Les premières solutions renvoyées par OrientedSearch sont déjà très diversifiées. Cependant, lorsque de nombreux motifs sont souhaités, le calcul du score dans OrientedSearch peut devenir trop coûteux. Dans ce cas, RandomSearch offre une grande diversité et constitue souvent l'approche la plus rapide.

D.4 Feature Models

Il est très important de tester efficacement les lignes de produits pour évaluer la qualité ou (dans le cas des lignes de produits logiciels) l'absence de bogues [138]. Dans les systèmes hautement configurables, cette tâche de test est compliquée par le grand nombre de caractéristiques en interaction. Par exemple, le noyau Linux contient des milliers de caractéristiques en interaction (telles que les options de compilation ou les bibliothèques installées) [140]. Les configurations (c'est-à-dire les ensembles de fonctionnalités) peuvent être testées en les instanciant sur la ligne de produits donnée (par exemple en compilant le noyau Linux avec des options et des bibliothèques spécifiques). Ces tests peuvent être coû-

teux (en termes de temps d'exécution [140], de mémoire [130], ou de main d'œuvre [127]), de sorte que des jeux de tests efficaces (un ensemble de configurations) doivent être générées.

Une façon de mesurer la qualité d'une suite de tests est la couverture t-wise [137]. Elle vise à garantir que toutes les interactions (combinaisons) d'un maximum de t caractéristiques sont testées. Mais il peut y avoir $2^t\binom{n}{t}$ t-wise combinaisons sur n caractéristiques. Ainsi, avec des milliers de caractéristiques, le calcul des combinaisons t-wise autorisées par la ligne de produit peut être prohibitif, sans parler de la génération d'un jeu de tests minimale qui couvre toutes ces combinaisons. Pour surmonter ce problème, des approches ont été développées qui utilisent des approximations basées sur des processus aléatoires tels que l'échantillonnage uniforme [84] ou pondéré [123]. Ces approches perdent les garanties, mais la diversité induite par le caractère aléatoire permet une bonne couverture expérimentale et de bons temps d'exécution.

Dans cette thèse, nous utilisons les stratégies de recherche aléatoire de la programmation par contraintes pour trouver des jeux de tests à couverture élevée. Les stratégies de recherche sont un moyen de faire en sorte que la recherche trouve des solutions dans différents espaces de solution. En particulier, les stratégies de recherche aléatoire n'ont pas besoin de calculer des métriques coûteuses (telles que le nombre de combinaisons autorisées) et peuvent générer des solutions diverses (c'est-à-dire à couverture élevée). Les contributions de ce chapitre sont les suivantes.

- Nous analysons les propriétés théoriques de la stratégie de recherche aléatoire par défaut, RandomSearch. Nous montrons que la distribution (non uniforme) des solutions renvoyées par cette stratégie est bien adaptée à la tâche de calcul d'un jeu de tests avec une bonne couverture t-wise.
- Nous concevons une amélioration de cette stratégie de recherche en utilisant des informations sur la ligne de produits : la fréquence des caractéristiques. La fréquence d'une caractéristique est le nombre de fois où elle apparaît dans toutes les configurations possibles. Nous utilisons cette information pour faire de meilleurs choix lors des décisions de la stratégie de recherche, afin de trouver des solutions qui couvrent plus de combinaisons inédites.

Nous expérimentons ces deux stratégies de recherche et les comparons aux approches d'échantillonnage les plus récentes. Nous montrons que les stratégies de recherche sont plus performantes que toutes les autres approches en termes de couverture et de temps d'exécution. Notre nouvelle approche améliore la stratégie de recherche aléatoire par dé-

faut sans aucun surcoût en termes de temps d'exécution. Nous montrons aussi qu'un échantillonnage uniforme est en fait préjudiciable à la couverture t-wise.

D.5 Diversité dans les Solveurs

La diversité comble le fossé entre les solveurs, avec la recherche rigide *backtrack-search*, et les utilisateur · ices, qui souhaitent se voir présenter plusieurs solutions différentes. Dans ce chapitre, nous considérons la diversité sous trois angles : comment les utilisateur · ices définissent leurs problèmes de diversité, comment les contraintes de diversité peuvent être implémentées dans les solveurs, et quelles sont les propriétés des algorithmes d'approximation.

Initialement, en PPC, la diversité a été définie dans [24]. Les auteurs définissent les problèmes `MaxDiverseKSet` et `MostDistant`, ainsi que les contraintes de diversité pour résoudre ces problèmes. Ces contraintes constituent une limite à la distance minimale entre toutes les solutions. Cependant, pour agréger les distances, la somme est souvent utilisée [44, 54]. Dans cette thèse, nous revenons sur les définitions de la diversité dans les problèmes contraints, et nous analysons en détail les définitions et les propriétés des contraintes de diversité.

Nous proposons des implémentations et prouvons les propriétés des contraintes, en fonction de l'agrégateur et de la distance utilisée. Nous prouvons une limite d'approximation pour l'approche gourmande. Nous analysons également le comportement moyen d'un échantillonneur uniforme dans l'hypercube unitaire, c'est-à-dire la diversité des solutions lorsqu'elles sont sélectionnées au hasard.

D.6 Multi-objectif

Dans les chapitres précédents, nous nous sommes concentrés sur les problèmes de satisfaction. Dans les problèmes d'optimisation à objectif unique, lorsque les solutions sont classées, l'utilisateur souhaite soit une solution unique (la solution optimale), soit quelques solutions proches de la solution optimale (en limitant la valeur de l'objectif). Il s'agit en fait d'un problème de satisfaction. Cependant, lorsque les objectifs sont multiples, les enjeux ne sont pas les mêmes.

Dans un problème à objectifs multiples, même si les solutions sont bien définies par des contraintes, il est plus difficile de trouver une solution optimale. Les objectifs peuvent

être contradictoires, de sorte que les solutions ne peuvent pas être comparées. Dans les problèmes de satisfaction, la diversité a été définie entre les instanciation des solutions (les valeurs réelles des variables). Dans les problèmes multi-objectifs, cependant, les solutions sont d'abord comparées en fonction de leurs valeurs objectives. Les utilisateurs doivent comprendre comment les objectifs interagissent et quelles sont les valeurs possibles des objectifs avant de prendre une décision. De cette manière, la diversité n'est pas définie sur les solutions, mais plutôt dans l'espace objectif.

Dans cette thèse, nous proposons une approche inspirée de PostHoc [28] pour trouver un bon ensemble de solutions diverses à présenter à un utilisateur. Cette approche en deux étapes consiste d'abord à trouver des solutions au problème, puis à en extraire un sous-ensemble de solutions diverses. Pour les deux étapes, nous présentons des approches état-de-l'art et nous concevons également de nouveaux algorithmes. Nous présentons un nouvel algorithme pour générer des points divers dans le simplexe, inspiré de l'algorithme de Lloyd. Nous présentons également une nouvelle méta-stratégie de recherche, Wavering, conçue pour les problèmes multi-objectifs. Nous montrons ensuite comment extraire des solutions du front de Pareto afin de les présenter à un utilisateur.

Conclusion

Dans cette thèse, nous avons montré que le hasard est un outil très puissant. Cela peut effrayer les utilisateurs au début, puisque par définition il n'y a pas de garanties fortes, mais nous avons montré qu'en moyenne il y a de nombreuses garanties. En outre, l'absence de garanties fortes peut être surmontée en appliquant une étape de post-traitement aux solutions générées aléatoirement. Cela permet d'obtenir des ensembles de solutions très diversifiés beaucoup plus rapidement que la recherche exhaustive, qui n'est généralement même pas applicable.

La diversité devrait être appliquée chaque fois que des décisions réelles sont prises. Cela signifie que les approches de diversité doivent être étendues à autant d'applications que possible. Dans cette thèse, nous avons toujours adopté des approches génériques (en utilisant le modèle comme une boîte noire), mais nous avons également utilisé la connaissance du domaine pour améliorer la recherche des solutions (comme la fréquence des caractéristiques dans les *feature models*). Il serait intéressant de développer un cadre qui permette aux utilisateurs de spécifier leur problème, mais aussi les propriétés souhaitées des solutions (diverses, couvrantes, optimales), sans tenir compte de la mise en

œuvre (contraintes de diversité, recherche aléatoire, optimisation de Pareto). Il ne s'agit pas seulement d'une API au-dessus d'un solveur de CP, il faudrait pouvoir modéliser la connaissance du domaine de manière générique et l'ajouter au processus de résolution de CP.

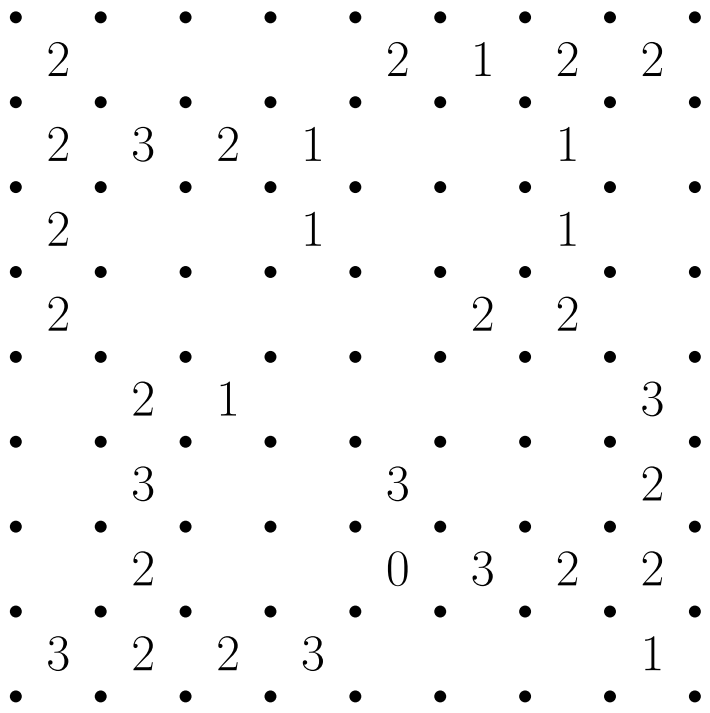


FIGURE D.1 – A *Slitherlink* grid.

COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

Nantes Université

Titre : Trouver des Solutions Diverses en Programmation par Contraintes avec des Approches Probabilistes

Mot clés : Programmation par Contraintes, Échantillonnage, Approche Probabiliste, Diversité, Solution

Résumé : Dans cette thèse, je présente de nouvelles approches pour générer des solutions aléatoires ou diverses dans le cadre de la Programmation Par Contraintes (PPC). Utilisées comme outil d'aide à la décision, les solutions impactent les personnes : la planification d'employé·es, l'itinéraire des livreur·euses, les congés des soignant·es de garde. L'algorithme utilisé dans les solveurs de PPC est efficace, mais c'est un cadre rigide, qui renvoie des solutions basées sur des heuristiques de branchement qui peuvent être biaisées en faveur d'un espace de solution particulier. Les décideur·euses veulent aussi choisir entre plusieurs solutions, ces solutions doivent donc être diversifiées.

Mon travail s'appuie sur des outils probabilistes. Le hasard est utilisé pour briser la rigidité du *backtrack-search* des solveurs de PPC et pour trouver des solutions dans un ordre différent à présenter à l'utilisateur·ice. Pour ce faire, j'ai conçu TABLESAMPLING, un échantillonneur travaillant dans le cadre de la PPC, qui bénéficie ainsi de toutes les améliorations des solveurs de PPC (temps d'exécution, ou nouvelles contraintes). Cependant, le caractère aléatoire n'est pas suffisant pour assurer la diversité. J'ai étudié et modifié des stratégies de recherche aléatoire pour générer des solutions diverses. La recherche peut ainsi être guidée vers des solutions dans des espaces intéressants.

Title: Finding Diverse Solutions in Constraint Programming with Probabilistic Approaches

Keywords: Constraint Programming, Sampling, Probabilistic Approach, Diversity, Solution

Abstract: In this thesis, I present new approaches to generate random or diverse solutions in the Constraint Programming (CP) framework. When used as a decision support tool, the solutions have an impact on people: the scheduling of employees, the route of delivery drivers, the day off for healthcare workers on rosters. The backtrack-search algorithm used in CP solvers is efficient, but it is also a rigid framework, returning solutions based on branching heuristics that may be biased towards a particular solution space. Furthermore, decision makers may also want to choose between multiple solutions, so these solutions should be diverse.

My work relies on probabilistic tools. Randomness is used to break the rigid backtrack-search of CP solvers and find solutions in a different order to present to a user. To do so, I designed TABLESAMPLING, a sampler working in the CP framework, that thus benefits from all the improvements in CP solvers (running time, or new constraints). However, randomness alone is not sufficient to provide diversity. I studied and modified random search strategies to generate diverse solutions. The search can thus be guided to solutions in interesting spaces.