PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Improvement and Integration of Counting-Based Search Heuristics in Constraint Programming
Auteur: Author:	Samuel Gagnon
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gagnon, S. (2018). Improvement and Integration of Counting-Based Search Heuristics in Constraint Programming [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3156/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3156/
Directeurs de recherche: Advisors:	Gilles Pesant
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

IMPROVEMENT AND INTEGRATION OF COUNTING-BASED SEARCH HEURISTICS IN CONSTRAINT PROGRAMMING

SAMUEL GAGNON
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MAI 2018

© Samuel Gagnon, 2018.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPROVEMENT AND INTEGRATION OF COUNTING-BASED SEARCH HEURISTICS IN CONSTRAINT PROGRAMMING

présenté par: GAGNON Samuel
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. DAGENAIS Michel, Ph. D., président
M. PESANT Gilles, Ph. D., membre et directeur de recherche
M. ROUSSEAU Louis-Martin, Ph. D., membre

DEDICATION

To my parents,
for which education has always been an important value...

À mes parents,
pour qui l'éducation a toujours été une valeur importante...

ACKNOWLEDGEMENTS

I would like to thank the Natural Sciences and Engineering Research Council of Canada for their financial support. I would also like to thank Gilles, my supervisor, which patiently guided me throughout my master's degree by encouraging me to try out my ideas.

J'aimerais remercier le Conseil de recherches en sciences naturelles et en génie du Canada d'avoir fourni un support financier à notre recherche. J'aimerais également remercier Gilles, mon directeur de recherche, qui m'a patiemment guidé tout au long de ma maîtrise en m'encourageant à essayer mes idées.

RÉSUMÉ

Ce mémoire s'intéresse à la *programmation par contraintes*, un paradigme pour résoudre des problèmes combinatoires. Pour la plupart des problèmes, trouver une solution n'est pas possible si on se limite à des mécanismes d'inférence logique; l'exploration d'un espace des solutions à l'aide d'heuristiques de recherche est nécessaire. Des nombreuses heuristiques existantes, les heuristiques de branchement basées sur le dénombrement seront au centre de ce mémoire. Cette approche repose sur l'utilisation d'algorithmes pour estimer le nombre de solutions des contraintes individuelles d'un *problème de satisfaction de contraintes*.

Notre contribution se résume principalement à l'amélioration de deux algorithmes de dénombrement pour les contraintes *alldifferent* et *spanningTree*; ces contraintes peuvent exprimer de nombreux problèmes de satisfaction, et sont par le fait même essentielles à nos heuristiques de branchement.

Notre travail fait également l'objet d'une contribution à un solveur de programmation par contraintes *open-source*. Ainsi, l'ensemble de ce mémoire est motivé par cette considération pratique; nos algorithmes doivent être accessibles et performants.

Finalement, nous explorons deux techniques applicables à l'ensemble de nos heuristiques: une technique qui réutilise des calculs précédemment faits dans l'arbre de recherche ainsi qu'une manière d'apprendre de nouvelles heuristiques de branchement pour un problème.

ABSTRACT

This thesis concerns constraint programming, a paradigm for solving combinatorial problems. The focus is on the mechanism involved in making hypotheses and exploring the solution space towards satisfying solutions: search heuristics. Of interest to us is a specific family called *counting-based search*, an approach that uses algorithms to estimate the number of solutions of individual constraints in *constraint satisfaction problems* to guide search.

The improvements of two existing *counting algorithms* and the integration of counting-based search in a *constraint programming solver* are the two main contributions of this thesis. The first counting algorithm concerns the *alldifferent* constraint; the second one, the *spanningTree* constraint. Both constraints are useful for expressing many constraint satisfaction problems and thus are essential for counting-based search.

Practical matters are also central to this work; we integrated counting-based search in an open-source constraint programming solver called Gecode. In doing so, we bring this family of search heuristics to a wider audience; everything in this thesis is built upon this contribution.

Lastly, we also look at more general improvements to counting-based search with a method for trading computation time for accuracy, and a method for learning new counting-based search heuristics from past experiments.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ANNEXES xiii

CHAPTER 1 INTRODUCTION 1
1.1 CSP Formulation of the Magic Square Problem 1
1.1.1 Inference 3
1.1.2 Search 4
1.1.3 Consistency Levels for Constraints 6
1.2 Problem Under Study 7

CHAPTER 2 LITERATURE REVIEW 9
2.1 Survey of Generic Branching Heuristics 9
2.1.1 Fail First Principle 9
2.1.2 Impact-Based Search 10
2.1.3 The Weighted-Degree Heuristic 10
2.1.4 Activity-Based Search 10
2.1.5 Counting-Based Search 11
2.2 Effort for Reducing Computation in the Search Tree 13
2.3 Adaptive Branching Heuristics 14

CHAPTER 3 ACCELERATING COUNTING-BASED SEARCH 15
3.1 Alldifferent Constraints 15
3.1.1 Improved Algorithm 16

3.1.2 Computing Maximum Solution Densities Only . . . 17
3.2 Spanning Tree Constraints . . . 18
3.2.1 Faster Specialized Matrix Inversion . . . 19
3.2.2 Inverting Smaller Matrices Through Graph Contraction . . . 19
3.3 Avoiding Systematic Recomputation . . . 20

CHAPTER 4 PRACTICAL IMPLEMENTATION OF COUNTING-BASED SEARCH IN GECODE . . . 22
4.1 Basic Notions in Gecode . . . 22
4.1.1 Main Loop in Gecode . . . 23
4.1.2 Creation of the Model . . . 23
4.1.3 Creation of the Search Strategy . . . 25
4.1.4 Solving the Problem . . . 26
4.2 Basic Structure for Supporting Counting-Based Search in Gecode . . . 26
4.2.1 Unique IDs for Variables . . . 28
4.2.2 Methods for Mapping Domains to their Original Values . . . 29
4.2.3 Computing Solution Distributions for Propagators . . . 30
4.2.4 Accessing Variable Cardinalities Sum in Propagators . . . 30
4.3 Counting Algorithms . . . 31
4.4 Creating a Brancher Using Counting-Based Search . . . 31

CHAPTER 5 EMPIRICAL EVALUATION . . . 33
5.1 Impact of our Contributions . . . 33
5.1.1 Acceleration of `alldifferent` . . . 33
5.1.2 Acceleration of `spanningTree` . . . 34
5.1.3 Avoiding Systematic Recomputation . . . 35
5.2 Benchmarking our Implementation in Gecode . . . 36
5.2.1 Magic Square . . . 37
5.2.2 Quasigroup Completion with Holes Problem . . . 38
5.2.3 Langford Number Problem . . . 39

CHAPTER 6 BRANCHING HEURISTIC CREATION USING A LOGISTIC REGRESSION . . . 41
6.1 Problem Definition . . . 41
6.2 Experiment . . . 43
6.2.1 First Logistic Function . . . 43
6.2.2 Second Logistic Function . . . 44

CHAPTER 7 CONCLUSION 47
7.1 Discussion of Contributions 47
7.2 Limits and Constraints 48
7.3 Future Research 49

REFERENCES 50

ANNEXES 55

LIST OF TABLES

Table 6.1 Offline database with densities for multiple single solution instances of the same problem. 42
Table 6.2 Logistic function combining $h_2 \wedge h_5$ against both branching heuristic. 44
Table 6.3 Logistic function combining $h_2 \wedge h_3$ against both branching heuristic. 44

LIST OF FIGURES

Figure 1.1 Partially filled magic square instance 2
Figure 1.2 Model for magic square in Fig 1.1 3
Figure 1.3 $\mathcal{P}$ 4
Figure 1.4 Solution space splitting 4
Figure 1.5 $\mathcal{P} \wedge h_1$ 5
Figure 1.6 Solution space split after h_2 5
Figure 1.7 $\mathcal{P} \wedge h_1 \wedge h_2$ 6
Figure 1.8 $\mathcal{P} \wedge h_1 \wedge h_2 \wedge h_3$ 6
Figure 1.9 $\mathcal{P} \wedge h_1$ 7
Figure 3.1 Incidence matrix for `alldifferent`(x_1, x_2, x_3) with $D_1 = \{1, 2\}$, $D_2 = \{2, 3\}$ and $D_3 = \{1, 3\}$ 15
Figure 3.2 Graph and its Laplacian matrix with $\gamma = 2$, meaning we can get every edge density by inverting two submatrices, e.g. L^1 (with inverse M^1 shown) and L^3 19
Figure 3.3 Contraction following assignments $e(1, 2) = 1$ and $e(1, 4) = 1$, with 1 as the representative vertex in connected component $\{1, 2, 4\}$ 20
Figure 3.4 Connected component $\{1, 2, 4\}$ as a single vertex 20
Figure 4.1 Gecode model for the magic square problem in Fig. 1.2 24
Figure 4.2 Variables for model in Fig. 4.1 24
Figure 4.3 Constraints for model in Fig. 4.1 25
Figure 4.4 Brancher for model in Fig. 4.1 26
Figure 4.5 Modeling the magic square with a custom brancher using Gecode . . 28
Figure 4.6 Principal components in relation with `CBSBrancher` 32
Figure 5.1 Percentage of Quasigroup Completion instances solved w.r.t. time and number of failures 34
Figure 5.2 Percentage of Hamiltonian Path instances solved w.r.t. time and number of failures with a 5 minute cutoff 35
Figure 5.3 The effect of different recomputation ratios for the Quasigroup Completion Problem, using the same instances as in Section 5.1.1 36
Figure 5.4 The effect of different recomputation ratios for Hamiltonian Path Problem, using same instances as in Section 5.1.2 36
Figure 5.5 Percentage of magic square instances solved w.r.t. time and number of failures 38

Figure 5.6 Percentage of latin square instances solved w.r.t. time and number of failures 39
Figure 5.7 Percentage of langford instances solved w.r.t. time and number of failures 40
Figure 6.1 Logistic function as a new branching heuristic 43
Figure 6.2 Rotation of $t(h_2, h_3)$ around the mean value of h_2 and h_3 45
Figure 6.3 Landscape of performance for 90 logistic functions 46
Figure A.1 Generic template for a counting algorithm in `solndistrib` 57
Figure A.2 Generic template for `domainsizesum` 58
Figure B.1 Good and bad branching choices according to each branching heuristic 59

LIST OF ANNEXES

Annexe A NOTES ABOUT COUNTING ALGORITHMS IMPLEMENTATION 55
Annexe B FEATURES ANALYSIS FOR LOGISTIC REGRESSION 59

CHAPTER 1 INTRODUCTION

Constraint programming (CP) is a declarative paradigm that has proven useful for solving large combinatorial problems, particularly in the areas of scheduling and planning. Appearing in the eighties, this approach was influenced by various fields such as Artificial Intelligence, Programming Languages, Symbolic Computing and Computational Logic [Barták (2018)]. Constraint programming can be seen as a competitor to other mathematical programming approaches such as linear programming (LP) or integer programming (IP). As constraint programming is more niche, one could ask why it is worth using instead of other paradigms.

One of the biggest selling points of CP is that the formulation of a problem is simple and straightforward; models in constraint programming are described by high level concepts that are close to the original problem formulation. With the *Constraint Satisfaction Problem* (CSP) as one of the main abstractions, modeling is done by expressing constraints on variables. This is in contrast with other mathematical programming approaches where constraints are translated to mathematical equations. Reasoning about the model and designing search heuristics is much simpler as a result.

Constraint programming solvers offer a catalog that goes from binary inequalities to complex constraints like bin packing and Hamiltonian path. Typically, these *global* constraints can be reformulated with several simpler constraints (like we would do in LP/IP). However, global constraints are able to solve problems faster because they encapsulate the semantic of bigger pieces in the problem; thus enabling models to be very precise [van Hoeve and Katriel].

In a constraint programming solver, a CSP is solved by giving a model of the problem and a search strategy, thus offering a declarative programming framework: the user describes the structure of the problem without specifying how to solve it. Behind the scenes, the solver combines various algorithms to solve the problem according to the specifications — its model and the search strategy.

In the next sections, basic notions in constraint programming will be introduced by looking at an example: the Magic Square Problem.

1.1 CSP Formulation of the Magic Square Problem

A CSP is formally described by the following tuple:

$$\mathcal{P} = \langle X, D, C \rangle \tag{1.1}$$

where

- $X = \{x_1, \ldots, x_n\}$ is a finite set of variables.
- $D = \{D_1, \ldots, D_n\}$ is a finite set of domains such that $x_i \in D_i$.
- $C = \{C_1, \ldots, C_t\}$ is a finite set of constraints. A constraint C_i on variables $x_1, \ldots, x_k$ is a relation that restricts the Cartesian product of its variables to a subset $C_i \subseteq D_1 \times \cdots \times D_k$.

A solution to $\mathcal{P}$ is an assignment that satisfies all constraints. As previously said, the first step in CP is to give a description of the problem.

16	·	·	·
·	10	·	·
·	·	7	·
·	·	·	·

Figure 1.1 Partially filled magic square instance

Let's take the example of the magic square instance shown in Fig. 1.1. An order n magic square is a n by n matrix containing the numbers 1 to n^2, with each row, column and main diagonal equal to $n(n^2+1)/2$ (#19 in CSPLib [Walsh (2018)]). The magic square in Fig. 1.1 is defined by the following CSP:

$$
\begin{aligned}
&\mathcal{P} = \langle X, D, C \rangle \\
&X = \{x_1, \ldots, x_{16}\} \\
&D = \{D_1, \ldots, D_{16} \mid D_i = \{1, \ldots, 16\}\} \\
&C = \{ \\
&\qquad x_1 = 16,\ x_6 = 10,\ x_{11} = 7, \\
&\qquad x_i \neq x_j,\ \forall 1 \leq i < j \leq 16, \\
&\qquad x_{4i+1} + x_{4i+2} + x_{4i+3} + x_{4i+4} = 34,\ \forall i \in \{0, \ldots, 3\}, \\
&\qquad x_i + x_{i+4} + x_{i+8} + x_{i+12} = 34,\ \forall i \in \{1, \ldots, 4\}, \\
&\qquad x_1 + x_6 + x_{11} + x_{16} = 34, \\
&\qquad x_4 + x_7 + x_{10} + x_{13} = 34, \\
&\}
\end{aligned}
$$

Figure 1.2 Model for magic square in Fig 1.1

1.1.1 Inference

Given the previous CSP, the next task is to do *constraint propagation* on the given problem to obtain *local consistency* for all constraints (thus obtaining what we call a *stable* state). In other words, we deduce everything we can from the CSP formulation of our problem. For example, we can deduce that $x_{16} = 1$ because the sum of the diagonals must be equal to 34. For a constraint, propagation is the process of filtering incoherent values from the domain of its variables.

Obtaining a stable state is done by scheduling all constraints for propagation. Each time a constraint propagates, it may reschedule another one by modifying a variable in its scope. Eventually, we obtain a stable state when this process finishes and all constraints have local consistency. In a finite domain constraint programming solver, this process will always terminate because each new iteration has to prune variable domains.

In our example, constraint propagation brings us to the following stable state:

16	2 3 4 5 6 8 9 11 12 13	2 3 4 5 6 8 9 11 12 13	2 3 4 5 6 8 9 11 12 13
2 3 4 5 6 8 9 11 12 13	10	2 3 4 5 6 8 9 11 12 13 14 15	4 5 6 8 9 11 12 13 14 15
2 3 4 5 6 8 9 11 12 13	2 3 4 5 6 8 9 11 12 13 14 15	7	4 5 6 8 9 11 12 13 14 15
3 4 5 6 8 9 11 12 13 14	4 5 6 8 9 11 12 13 14 15	4 5 6 8 9 11 12 13 14 15	1

Figure 1.3 $\mathcal{P}$

Here, we could ask why $14 \in D_{13}$: there is no valid assignment satisfying $16+x_5+x_9+14 = 34$, as x_5 and x_9 should both be equal to 2, thus violating the `alldifferent` constraint. The key observation is that both constraints are locally consistent, the `alldifferent` constraint is not involved for evaluating supporting solutions of other `linear` constraints.

1.1.2 Search

Constraint propagation was not sufficient for finding a valid solution to our problem; we still have to find a valid assignment for 12 variables. To this end, we need the second component of the solver: a search strategy. The search strategy is used to make hypotheses to prune the search space when no more constraint propagation is possible. A valid search strategy for our problem could be as simple as:

- Select the variable x_i with the smallest domain
- Select the smallest value $v \in D_i$
- Split the solution space as $\mathcal{P} \wedge (x_i = v) \ \cup \ \mathcal{P} \wedge (x_i \neq v)$

This strategy splits $\mathcal{P}$ into two valid CSPs (with $h_1 : \ x_i = v$):

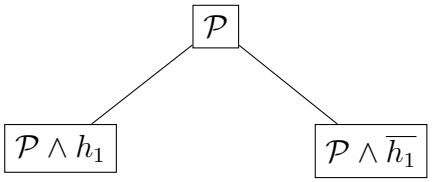


Figure 1.4 Solution space splitting

If we apply this search strategy to our example, we have x_5 with a cardinality of 10 and a minimum value of 2, giving us the hypothesis $h_1 : x_5 = 2$. The component of the search strategy that gives hypotheses is called a *branching heuristic*. After constraint propagation, we get the state shown in Fig. 1.5.

16	3 4 5 6 8 9 11	3 4 5 6 8 9 11	4 5 6 8 9 11 12
2	10	8 9 11 12 13 14	8 9 11 12 13 14
3 4 5 6 8 9 11 12 13	3 4 5 6 8 9 11 12 13 14 15	7	8 9 11 12 13 14 15
3 4 5 6 8 9 11 12 13	5 6 8 9 11 12 13 14 15	5 6 8 9 11 12 13 14 15	1

Figure 1.5 $\mathcal{P} \wedge h_1$

However, we still haven't found a solution. If we use our branching heuristic again, we get a second hypothesis $h_2 : x_7 = 8$ that leads, after constraint propagation, to the state in Fig. 1.7.

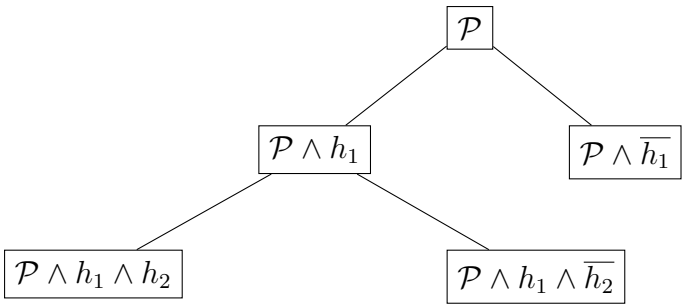


Figure 1.6 Solution space split after h_2

16	6 9	4 5 6	4 5 6
2	10	8	14
3 4 5	9 11	7	13 15
11 12 13	5 6 9	13 15	1

Figure 1.7 $\mathcal{P} \wedge h_1 \wedge h_2$

Finally, a third hypothesis $h_3 : x_2 = 6$ will lead us to a failed state. In other words, finding a solution is now impossible as the maximum sum we can obtain in the first row is $16 + 6 + 5 + 5 = 31$, which is lower than 34. This means that our sequence of hypotheses (shown in red) is incorrect.

16	6	4 5	4 5
2	10	8	14
3 4 5	9 11	7	13 15
11 12 13	5 9	13 15	1

Figure 1.8 $\mathcal{P} \wedge h_1 \wedge h_2 \wedge h_3$

If we continue the exploration of this implicit tree in a depth-first search fashion, we will eventually find a solution for this particular instance.

1.1.3 Consistency Levels for Constraints

When a constraint propagates to achieve local consistency, it may use different algorithms to do so. One way to classify those algorithms is by the *consistency level* they enforce. Applying different levels of consistency may yield different stable states.

In Fig. 1.5, constraint propagation is done by applying *bounds consistency* for all constraints after making hypothesis h_1. Doing so ensures that all variables have a supporting solution for the minimum and maximum value in their domain. In other words, for a given constraint and bound value, if it is impossible to assign other variables while satisfying the constraint, we can remove it.

If we instead apply *domain consistency* after making h_1, we get an alternative stable state to Fig. 1.5:

16	3 4 5 6 8 9 11	3 4 5 6 8 9 11	4 5 6 8 9 11 12
2	10	8 9 11 13 14	8 9 11 13 14
3 4 5 8 11 12 13	3 4 5 6 8 9 11 12 13 14 15	7	8 9 11 12 13 14 15
3 4 5 8 11 12 13	5 6 8 9 11 12 13 14 15	5 6 8 9 11 12 13 14 15	1

Figure 1.9 $\mathcal{P} \wedge h_1$

Domain consistency is stronger than bounds consistency; instead of only looking at bounds for incoherent values, we look at the whole domain. It takes more time, but more incoherent values are filtered as a result, as Fig. 1.9 shows with x_7, x_8, x_9, and x_{13}. Let's take the `linear` constraint of the first column: $16 + 2 + x_9 + x_{13} = 34$. If we assign $x_9 = 6$, there's no possible solution for x_{13} as $34 - 16 - 2 - 6 = 10 \notin D_{13}$. This incoherent value was not removed by the bounds consistency algorithm before.

1.2 Problem Under Study

In the previous example, a different search strategy may have led to a solution without a single backtrack; another one, to a solution after more backtracks. Making an incorrect hypothesis in the first node of the search tree can be very costly if we explore the search space in a regular backtracking fashion. Choosing a good search strategy is of great importance.

One problem in constraint programming is the lack of good generic search heuristics as opposed to linear or integer programming; when modeling a new problem, users are often required to make custom heuristics to obtain good performance. This thesis is interested

in *counting-based search* (CBS) [Zanarini and Pesant (2007)], a family of generic search heuristics in CP that uses *solution counting*. This approach exploits the expressiveness of CP while being easy to use for new problems. However, in its current state, it is mainly limited to academia. In this thesis, the following issues are addressed:

- Depending on the types of constraints present in the model computing a branching choice can be prohibitively slow compared to other branching heuristics.
- Counting-based search lacks visibility; it is not available in popular constraint programming library.
- Selecting the correct branching heuristic involves trial and error.

We propose the following contributions, with the first two published in a paper to be presented on the *15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research* under the name *Accelerating Counting-Based Search*:

- Computational improvements to *counting algorithms* for two constraints, namely `alldifferent` (Section 3.1) and `spanningTree` (Section 3.2).
- A generic method to avoid calling counting algorithms at every node of the search tree (Section 3.3).
- Support for CBS in open-source solver that is popular both with academia and industry (Section 4).
- An experiment for learning new branching heuristics based on previous observations (Section 6).

CHAPTER 2 LITERATURE REVIEW

2.1 Survey of Generic Branching Heuristics

Constraint programming builds concise models from high-level constraints that reveal much of the combinatorial structure of a problem. That structure is used to prune the search space through domain filtering algorithms, to guide its exploration through branching heuristics, and to learn from previous attempts at finding a solution.

Having good models in constraint programming often requires crafting custom branching heuristics. Other paradigms like Mixed Integer Programming (MIP) and SAT solvers are easier to use in this regard; they perform really well with default search strategies (Michel and Van Hentenryck (2012)). This success prompted the constraint programming community to design new generic heuristics in recent years. Having powerful generic heuristics lowers the complexity of modeling a problem in CP; one can focus on defining the CSP rather than describing how to search the solution space.

All presented branching heuristics follow a *dynamic ordering*, meaning they choose their next variable during search as opposed to a *static ordering*, which determines the order of variables for branching before starting search; dynamic ordering performs usually much better because it is more informed.

2.1.1 Fail First Principle

One of the first guiding principles for designing search strategies is given in [Haralick and Elliott (1980)] and is known as the *fail first principle*: the variable x_i involved in the next branching decision should be the one that is the most likely to lead to failure. While it may seem counter-intuitive, applying this strategy is the fastest way to prove that a sub-tree in the current search space has no solutions — if every possible value in D_i leads to failure, it is a proof that the current problem has no solution, and we can backtrack immediately. The fail first principle can be used to design various strategies for selecting variables in branching heuristics. As an example, we can select the variable that has the smallest domain (it is more constrained than other variables) or the variable involved in the most constraints; this principle has inspired a variety of variable selection strategies in branching heuristics.

2.1.2 Impact-Based Search

Impact-based search (IBS) is inspired by ideas originating from MIP solvers [Refalo (2004)]. Like its name implies, it defines the notion of *impact* for variables; a variable has a big impact if its assignment greatly reduces the search space by doing a lot of constraint propagation. Thus, exploring solutions by looking at variables with big impact leads to smaller search trees.

This strategy is generic because it looks at the reduction of the search space for the given instance. The size of the search space is approximated by:

$$P = |D_1| \times \cdots \times |D_n| \tag{2.1}$$

As a consequence, the impact of an assignment, with P_{before} and P_{after} denoting P before and after the variable assignment and the resulting constraint propagation, is defined as:

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}} \tag{2.2}$$

Equation 2.2 can then be used to define the impact of a variable. Impact branching heuristics are generic; they do not use special knowledge about the problem at hand and can readily be applied to all models.

2.1.3 The Weighted-Degree Heuristic

WDEG [Boussemart et al. (2004)] uses information about the previous nodes seen in the search tree: each time a constraint is violated during search, an associated weight is increased. This allows to give a *weighted-degree* to each variable by doing a weighted sum of every constraint it is part of. By focusing on variables with big weighted-degrees, one can design a branching heuristic following the fail-first principle.

2.1.4 Activity-Based Search

Like IBS, the activity-based seach (ABS) [Michel and Van Hentenryck (2012)] is influenced by a heuristic from a solver with a different paradigm: the SAT heuristic VSIDS [Moskewicz et al. (2001)].

ABS, like WDEG, associates a counter to each variable. However, the counter instead keeps track of every time the domain of a variable is pruned, thus giving a measure of *activity* for each variable. This counter decays during search, hence slowly forgetting about past activity.

This information can be used to select variables with the highest activity.

2.1.5 Counting-Based Search

Counting-based search (CBS) [Zanarini and Pesant (2007)] represents a family of branching heuristics that guides the search for solutions by identifying likely variable-value assignments in each constraint. Given a constraint $c(x_1, \ldots, x_n)$, its number of solutions $\#c(x_1, \ldots, x_n)$, respective finite domains $D_{i\ 1 \leq i \leq n}$, a variable x_i in the scope of c, and a value $v \in D_i$, we call

$$\sigma(x_i, v, c) = \frac{\#c(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n)}{\#c(x_1, \ldots, x_n)} \tag{2.3}$$

the *solution density* of pair (x_i, v) in c, i.e. how often a certain assignment is part of a solution to c.

Let's suppose we have the constraint $c_i = \texttt{alldifferent}(x_1, x_2, x_3)$ with $D_1 = \{1, 2\}$, $D_2 = \{2, 3\}$ and $D_3 = \{1, 2, 3\}$. We can verify by hand that this constraint admits 3 solutions. If we fix $x_1 = 1$, we are left with $D_2 = \{2, 3\}$, $D_3 = \{2, 3\}$, and 2 possible solutions. Thus, we say that the solution density of the assignment $x_1 = 1$ is $\frac{2}{3}$; it reduces the search space by 33% for this constraint.

CBS was originally inspired by IBS [Pesant (2005)]. The idea was to count solutions for individual constraints, thus refining the notion of impact in IBS; given a constraint $c(x_1, \ldots, x_n)$, we know that $\#c(x_1, \ldots, x_n) \leq |D_1| \times \cdots \times |D_n|$. However, this approach relies on having specialized *counting algorithms* for constraints. To this effect, the original paper also gave an idea of how solution counts could be computed for several constraints.

One of the core ideas of counting-based search is to exploit the constraints of the model when branching. While other heuristics like IBS and ABS consider constraint propagation, they do not take into account constraint types, thus missing on the expressivity of constraint programming.

Generic Counting-Based Search Heuristic Framework

In [Zanarini and Pesant (2007)], a first framework for designing branching heuristics with solution counts was given. While it does not exactly correspond to the notation in the article, the general principle is given in Alg. 1. For each constraint, we compute the solution density of every $\langle variable, value \rangle$ pair and put it in SD. Afterwards, a procedure — here called `CBS_HEUR` — can aggregate solution densities and propose an *integrated* branching choice

(i.e. the procedure returns both the variable and the value instead of just the variable).

```
1 SD = {}
2 foreach constraint c(x_1, ..., x_n) do
3     foreach unbound variable x_i ∈ {x_1, ..., x_n} do
4         foreach value d ∈ D_i do
5             SD[c][x_i][d] = σ(x_i, d, c)
6 return CBS_HEUR(SD)                    ▷ Return a branching decision using SD
```

Algorithm 1: Generic counting-based search generic decision flow, adapted from [Zanarini and Pesant (2007)]

`CBS_HEUR` can be a variety of branching heuristics, as shown by [Zanarini (2010)]. One simple combination that works well in practice, called *maxSD*, branches on $x_i^\star = v^\star$ where

$$(x_i^\star, v^\star, c^\star) = \underset{c(x_1,\ldots,x_n)\in C,\, i\in\{1,\ldots,n\},\, v\in D_i}{\operatorname{argmax}} \sigma(x_i, v, c) \tag{2.4}$$

Another one, called *aAvgSD*, also branches on $x_i^\star = v^\star$:

$$(x_i^\star, v^\star) = \underset{i\in\{1,\ldots,n\},\, v\in D_i}{\operatorname{argmax}} \frac{\sum_{c\in\Gamma(x_i)} \sigma(x_i, v, c)}{|\Gamma(x_i)|} \tag{2.5}$$

where $\Gamma(x_i)$ is the set of constraints in which x_i is included. Both heuristics branch on $x_i^\star \neq v^\star$ upon backtracking.

Counting Algorithms

Between 2005 and 2012, various papers were published to propose counting algorithms for different constraints:

- `regular` constraint (as in regular language) [Pesant and Quimper (2008)];
- `knapsack` constraint [Pesant and Quimper (2008)];
- `alldifferent` constraint [Zanarini and Pesant (2010)];
- `element` constraint [Pesant and Zanarini (2011)];
- `gcc` or global cardinality constraint [Pesant et al. (2012)].

In [Pesant and Quimper (2008)], the authors benchmark with good results different counting-based search heuristics against other generic methods such as IBS and WDEG on the following problems:

- Quasigroup Completion Problem with Holes (QCP)
- Magic Square Completion Problem
- Nonograms
- Multi Dimensional Knapsack Problem
- Market Split Problem
- Rostering Problem
- Cost-Constrained Rostering Problem
- Traveling Tournament Problem with Predefined Venues (TTPPV)

The previous list shows that only a handful of constraints can already be used to model a lot of problems. In later research, the following counting algorithms were also introduced:

- `spanningTree` constraint [Brockbank et al. (2013)];
- `spread`/`deviation` constraints [Pesant (2015)];
- minimum weighted adaptation for the `spanningTree` constraint [Delaite and Pesant (2017)].

2.2 Effort for Reducing Computation in the Search Tree

Given we have a search heuristic that performs well, constraint propagation is going to be the next decisive factor for the performance of the model. It will mainly depend on the constraints used for modeling — a problem may admit different models — and their consistency levels.

Choosing the right constraints and consistency levels for a problem is a difficult task. While important for propagation, it will also influence how CBS performs; counting heuristics are specialized for each constraint and counting for a given constraint may be harder than for another one. Moreover, the same constraint can have different counting algorithms depending on the consistency level. As an example, `knapsack` has a different algorithm for domain

consistency and bounds consistency, with the latter version being easier to compute [Pesant et al. (2012)].

The task of automating the choice of consistency has received some attention in the constraint programming community in recent years.

Static analysis on finite domain constraint logic programs has been used to decide when a constraint with domain consistency could be replaced with another with bounds consistency, while keeping the search space identical [Schulte and Stuckey (2001)]. In [Schulte and Stuckey (2008)], a dynamic version with the same guarantee was designed and tested in a CP solver.

Heuristic approaches were also tried for tackling the similar problem of dynamically switching between weak and strong local consistencies for binary CSPs [Stergiou (2008)], and for non-binary CSPs in [Paparrizou and Stergiou (2012)] and [Woodward et al. (2014)]. The latter approach counts support for variable-value pairs, meaning they use information similar to CBS.

2.3 Adaptive Branching Heuristics

In constraint programming, computing branching choices is usually very fast (with counting-based search and IBS being notable exceptions). As a result, excluding the design of new branching heuristics (both generic or specialized), techniques for automating the choice or design of branching heuristics are a big research topic. In [Burke et al. (2013)], such techniques are described as *hyper-heuristics*. Broadly speaking, hyper-heuristics can be classified in two categories: heuristics for selecting other heuristics or heuristics for generating heuristics.

[Terashima-Marín et al. (2008)] and [Ortiz-Bayliss et al. (2012)] both proposed a high level heuristic that learns to choose low-level heuristics depending on the problem state. [Ortiz-Bayliss et al. (2010)] also investigated how two different variable ordering heuristics performed depending on constraint density and tightness, and this information was used in making a hyper-heuristic using both variable ordering strategies. All these techniques were used on constraint satisfaction problems, but did not use CP.

There are also other approaches that do not characterize themselves as hyper-heuristics. In [Epstein et al. (2002)], an architecture for combining the expertise of different competing heuristics, here called *Advisors*, was adapted from an old framework called *FORR* [Epstein (1994)]. Later, [Soto et al. (2012)] also explored the problem of choosing the right combination of heuristics for a given problem.

CHAPTER 3 ACCELERATING COUNTING-BASED SEARCH

The cost of computing solution densities depends on the constraint: for some it is only marginally more expensive than its existing filtering algorithm (e.g. `regular`) while for others exact computation is intractable (e.g. `alldifferent`). As stated in the introduction, the extra work done by counting algorithms is often a bottleneck when solving a model, leading to slower execution time.

This chapter presents several contributions to accelerate counting-based search. We first discuss specific improvements for the `alldifferent` and `spanningTree` counting algorithms in Section 3.1 and 3.2. Then a generic method for accelerating search is presented in Section 3.3. All discussed algorithms are implemented using Gecode [Gecode Team (2017)] and available in [Gagnon (2017)].

3.1 Alldifferent Constraints

An instance of an `alldifferent`$(x_1, \ldots, x_n)$ constraint is equivalently represented by an incidence matrix $A = (a_{iv})$ with $a_{iv} = 1$ whenever $v \in D_i$ and $a_{iv} = 0$ otherwise. For notational convenience and without loss of generality, we identify domain values with consecutive natural numbers. Because we will want A to be square (with $m = |\bigcup_{x_i \in X} D_i|$ rows and columns), if there are fewer variables than values we add enough rows, say p, filled with 1s. It is known that counting the number of solutions to the `alldifferent` constraint is equivalent to computing the *permanent* of that square matrix (dividing the result by $p!$ to account for the extra rows) [Zanarini and Pesant (2010)]:

$$perm(A) = \sum_{v=1}^{m} a_{1v}\, perm(A^{1v}) \tag{3.1}$$

where A^{ij} denotes the sub-matrix obtained from A by removing row i and column j.

$$A = \left(\begin{array}{c|ccc} & v=1 & v=2 & v=3 \\ \hline x_1 & 1 & 1 & 0 \\ x_2 & 0 & 1 & 1 \\ x_3 & 1 & 0 & 1 \end{array}\right)$$

Figure 3.1 Incidence matrix for `alldifferent`(x_1, x_2, x_3) with $D_1 = \{1, 2\}$, $D_2 = \{2, 3\}$ and $D_3 = \{1, 3\}$

Since computing the permanent is $\#P$-complete [Valiant (1979)], Zanarini and Pesant proposed approximate counting algorithms for the `alldifferent` constraint based on sampling [Zanarini and Pesant (2007)] and upper bounds [Pesant et al. (2012)]. Algorithm 2 reproduces the latter using notation adapted for this article. As each assignment $x_i = v$ in the `alldifferent` constraint induces a different incidence matrix, a naive approach to compute solution densities is to recompute the permanent upper bound for each assignment. However, our upper bounds are a product of factors F for each variable x_i which depend only on the size of its domain $d_i = |D_i|$ (line 1). Hence if we account for the domain reduction of the assigned variable (line 5) and of each variable which could have taken that value (line 6) — simulating forward checking — we can compute the solution density of each assignment (line 10) by updating the upper bound UB_A calculated for the whole constraint (line 1). Reusing UB_A avoids recomputing upper bounds from scratch. Let c_v denote the number of 1s in column v of A. Given that we can precompute the factors, the total computational effort is dominated by line 6 where we do a total of $\Theta(\sum_{v=1}^{m} c_v^2)$ operations: for a given value v, u_v is computed c_v times by multiplying $c_v - 1$ terms.

```
1  UB_A = ∏_{x_i} F[d_i]                                   ▷ Constraint upper bound
2  foreach x_i ∈ X do
3      total = 0                                           ▷ Normalization factor
4      foreach v ∈ D_i do
5          u_{x_i} = F[1] / F[d_i]                         ▷ Variable assignment update
6          u_v = ∏_{k≠i : v∈D_k} F[d_k − 1] / F[d_k]       ▷ Value assignment update
7          UB_{x_i=v} = UB_A · u_{x_i} · u_v               ▷ Assignment upper bound
8          total += UB_{x_i=v}
9      foreach v ∈ D_i do
10         SD[i][v] = UB_{x_i=v} / total
11 return SD
```

Algorithm 2: Solution densities for `alldifferent`, adapted from [Pesant et al. (2012)]

3.1.1 Improved Algorithm

The product at line 6 of Algorithm 2 can be rewritten to depend only on v:

$$\mathrm{u}_v = \frac{F[d_i]}{F[d_i - 1]} \prod_{k \,:\, v \in D_k} \frac{F[d_k - 1]}{F[d_k]} \quad . \tag{3.2}$$

This allows us, as shown in Algorithm 3, to precompute this product for every value (line 1-4) as it does not depend on i anymore, leading to each $\mathrm{UB}_{x_i=v}$ being computed in constant time

(line 8). We also avoid computing UB_A since that factor cancels out during normalization. Algorithm 3 runs in $\Theta(\sum_{v=1}^{m} c_v)$ time, which is asymptotically optimal if we need to compute every solution density (since $\sum_{v=1}^{m} c_v = \sum_{i=1}^{n} d_i$).

```
1  UB_v = 1, ∀v ∈ {1, 2, ..., m}
2  foreach x_i ∈ X do
3  |   foreach v ∈ D_i do
4  |   |   UB_v *= F[d_i − 1] / F[d_i]
5  foreach x_i ∈ X do
6  |   total = 0
7  |   foreach v ∈ D_i do
8  |   |   UB_{x_i=v} = F[1] / F[d_i − 1] · UB_v
9  |   |   total += UB_{x_i=v}
10 |   foreach v ∈ D_i do
11 |   |   SD[i][v] = UB_{x_i=v} / total
12 return SD
```

Algorithm 3: Improved version of Algorithm 2.

3.1.2 Computing Maximum Solution Densities Only

Some search heuristics, such as *maxSD*, only really need the highest solution density from each constraint in order to make a branching decision. In such a case it may be possible to accelerate the counting algorithm further. We present such an acceleration for the `alldifferent` constraint.

The factors F in our upper bounds are strictly increasing functions, meaning that for a given value v, the highest solution density will occur for the variable with the smallest domain. Algorithm 4 identifies that peak for each value, knowing that the highest one will be included in this subset. Note however that because we don't compute a solution density for each value in the domain of a given variable, we cannot normalize them as before (though we at least adjust for the p extra rows). So we may lose some accuracy but what we were computing was already an estimate, not the exact density. The asymptotic complexity of this algorithm remains the same as the previous one, but makes fewer computations: we iterate on each

variable and value once instead of three times.

$\mathrm{UB}_v = (\frac{F[n-1]}{F[n]})^p$, $\min_v = 1,\ \forall v \in \{1, 2, \ldots, m\}$
foreach $x_i \in X$ **do**
  foreach $v \in D_i$ **do**
    UB_v *$= \frac{F[d_i - 1]}{F[d_i]}$
    if $d_i < d_{\min_v}$ **then**
      $\min_v = i$
maxSD $= \{var = 0,\ val = 0,\ dens = 0\}$
foreach $v \in \{1, 2, \ldots, m\}$ **do**
  $\mathrm{SD}[\min_v][v] = \frac{F[1]}{F[d_{\min_v} - 1]} \cdot \mathrm{UB}_v$
  if $\mathrm{SD}[\min_v][v] >$ maxSD.$dens$ **then**
    maxSD $= \{\min_v,\ v,\ \mathrm{SD}[\min_v][v]\}$
return maxSD

Algorithm 4: Maximum solution density for `alldifferent`

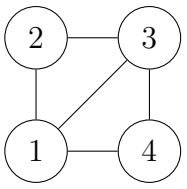
An evaluation of improvements of Algorithm 2, 3 and 4 is given in Section 5.1.1.

3.2 Spanning Tree Constraints

Brockbank, Pesant and Rousseau introduced an algorithm to compute solution densities for the `spanningTree` constraint in [Brockbank et al. (2013)]. The graph is represented as a *Laplacian matrix L* (vertex degrees on the diagonal and edges indicated by -1 entries) and Kirchhoff's Matrix-Tree Theorem [Chaiken and Kleitman (1978)] is used to compute solution densities for every edge (u, v) using the following formula:

$$\sigma((u, v), 1, \texttt{spanningTree}(G, T)) = m^u_{v'v'} \tag{3.3}$$

with $M^u = (m^u_{ij})$ defined as the inverse of the sub-matrix L^u obtained by removing row and column u from L and v' equal to v if $v < u$ and to $v - 1$ otherwise. Given a vertex cover of size γ on a graph over n nodes, computing all solution densities takes $\mathcal{O}(\gamma n^3)$ time. Figure 3.2 shows an example graph and its Laplacian matrix.



$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \quad M^1 = \begin{pmatrix} 5/8 & 2/8 & 1/8 \\ 2/8 & 4/8 & 2/8 \\ 1/8 & 2/8 & 5/8 \end{pmatrix}$$

Figure 3.2 Graph and its Laplacian matrix with $\gamma = 2$, meaning we can get every edge density by inverting two submatrices, e.g. L^1 (with inverse M^1 shown) and L^3

With this formula counting-based search heuristics can be used on problems such as degree constrained spanning trees (and Hamiltonian paths in particular) with very good results [Brockbank et al. (2013)]. However, they become impractical for large instances because of repeated matrix inversion. The following sections address this problem by proposing two improvements. Note that these improvements remain valid for the recent generalization to weighted spanning trees [Delaite and Pesant (2017)].

3.2.1 Faster Specialized Matrix Inversion

By construction, the sub-matrix L^u we invert has a special form that enables us to use a specialized algorithm. It is Hermitian (more precisely, integer symmetric). Since the row and column removed from L have the same index u, it is *diagonally dominant*: $|\ell^u_{ii}| \geq \sum_{j \neq i} |\ell_{ij}|, \forall i$. Its diagonal entries are positive. Therefore it is positive semidefinite or, equivalently, has non-negative eigenvalues. The Matrix-Tree Theorem states that the number of spanning trees is equal to the determinant of L^u, itself equal to the product of its eigenvalues. Therefore each eigenvalue is strictly positive and L^u is positive definite.

A Hermitian positive definite matrix can be inverted via Cholesky factorization instead of the standard LU factorization. Inverting a positive definite matrix requires approximately $\frac{1}{3}n^3$ (Cholesky factorization) + $\frac{2}{3}n^3$ floating-point operations whereas inverting a general matrix requires approximately $\frac{2}{3}n^3$ (LU factorization) + $\frac{4}{3}n^3$ floating-point operations [Choi et al. (1996)]. We therefore expect a two-fold improvement in runtime.

3.2.2 Inverting Smaller Matrices Through Graph Contraction

When branching, if an edge (u, v) is fixed, the Laplacian matrix must be updated to reflect this change. The technique described by Brockbank, Pesant and Rousseau is the following: if it is forbidden, we set $l_{uv} = 0$; if it is required, we must contract it in the graph, meaning we transfer the edges of vertices u and v to a representative vertex in the same connected

component while keeping a 1 on the diagonal of these vertices to keep the matrix invertible. Figure 3.3 shows an example.

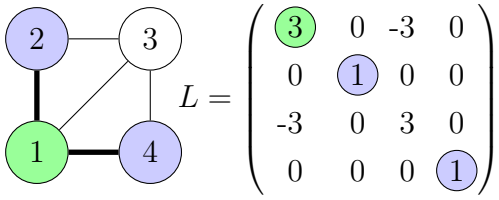


Figure 3.3 Contraction following assignments $e(1,2) = 1$ and $e(1,4) = 1$, with 1 as the representative vertex in connected component $\{1,2,4\}$

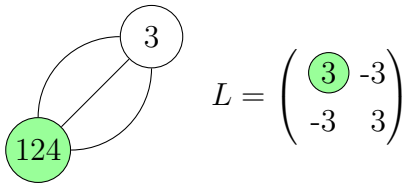


Figure 3.4 Connected component $\{1,2,4\}$ as a single vertex

This way of updating L works but still requires that we invert $(n-1) \times (n-1)$ matrices to compute solution densities throughout the search. However, as shown in Fig. 3.4, we can view each connected component as a single vertex, leading to smaller Laplacian matrices, and thus smaller matrices to invert as we fix edges.

An evaluation of improvements of Section 3.2.1 and 3.2.2 is given in Section 5.1.2.

3.3 Avoiding Systematic Recomputation

The improvements we presented so far are specific to the `alldifferent` and `spanningTree` constraints. In this section we present an additional technique applicable to any constraint in order to avoid recomputation but at the expense of accuracy. Usually at every node of the search tree, before branching, we systematically call the counting algorithm for each constraint. Suppose we have a `spanningTree` constraint on a graph with hundreds of vertices and thousands of edges: we may have fixed a single edge with very few changes propagated since the last call to its counting algorithm but the whole computation, involving the expensive inversion of large matrices, will be undertaken again even though the resulting solution densities are likely to be very similar. To avoid this we propose a simple dynamic technique: while the variable domains involved remain about the same, we do not recompute solution densities for a constraint but use the latest ones as an estimate instead. For any given node k in the search tree and constraint c, let $S_c^k = \sum_{x_i \in c} d_i$. We recompute only if $S_c^k \leq \rho S_c^j$, with $0 < \rho \leq 1$ some appropriate ratio and j the last node above k but on the same path

from the root for which we computed solution densities for c. Note that as opposed to a static criterion such as calling the counting algorithm of every constraint at fixed intervals of depth in the search tree, our approach adapts dynamically to individual constraints and to how quickly the domains of the variables in their scope shrink.

An evaluation of this method is given in Section 5.1.3.

CHAPTER 4 PRACTICAL IMPLEMENTATION OF COUNTING-BASED SEARCH IN GECODE

At the start of this Master's degree, counting-based search heuristics were only available in a private fork of ILOG Solver [IBM (2018)], a commercial constraint programming solver. This made it difficult for other researchers to test our heuristics, hence leading the slower adoption among academia.

Therefore, introducing counting-based search heuristics in a more readily available solver was the first logical step to undertake. For this task, we chose Gecode for the following reasons:

- State-of-the-art performance
- Feature-rich
- Good documentation
- Free and open-source

Furthermore, from the perspective of a graduate student, modifying Gecode led to a better understanding of constraint programming — both in theory and practice, and made it easier to implement ideas that were central to the other half of this work.

Section 4.1 introduces a basic vocabulary and theoretical setting for understanding Gecode and the following subsections. Section 4.2 explains the required changes to Gecode to support counting-based search. In Section 4.3 we give a quick overview of the counting algorithm we implemented, thus making it possible to design a brancher using counting-based search (Section 4.4).

4.1 Basic Notions in Gecode

This section is not a full introduction to the real inner working of the solver; with close to 2 000 classes and 300 000 lines of C++, Gecode is very complex and such an introduction would not be useful for our purpose.

Instead, we will work under various simplifying assumptions to make it easier to introduce just enough Gecode terminology for understanding the next sections.

All the material in this section will be introduced alongside the same practical problem as in the introduction: solving the partially filled magic square in Fig. 1.1.

4.1.1 Main Loop in Gecode

Under the (naïve) assumption that we must return all solutions, that we always copy the problem state, that we only make binary branching choices (more on that later), and that we explore the search space in a depth-first search fashion, Alg. 5 is a good representation of Gecode's main loop.

```
1 Function SOLVE(space)
2 |   space' ← PROPAGATE(space)
3 |   if space' is invalid then
4 |   |   return ∅
5 |   if space' is valid then
6 |   |   return solution
7 |   s_1, s_2 ← BRANCH(space')
8 |   return SOLVE(s_1) ∪ SOLVE(s_2)
```

Algorithm 5: Solving a constraint programming problem in Gecode, adapted from [Tack (2009)]

According to this algorithm, the first step for solving our magic square is to create a space and pass it to this function. In Gecode, a space is any class inheriting the base class `Space` and it encapsulates the model and the search component of the problem. Before talking about this function, we will show how we create a space that follows the mathematical model given in Fig. 1.2.

4.1.2 Creation of the Model

Our new class will be called `MagicSquare` and the model has to be defined inside its constructor:

```
class MagicSquare : public Space {
protected:
    // Variables of the model
    IntVarArray x;
public:
    MagicSquare(void) {
        // 1. Initialization of variables
        // 2. Initialization of propagators
        // 3. Initialization of branchers
    }
    /* ...
     * ...
     */
};
```

Figure 4.1 Gecode model for the magic square problem in Fig. 1.2

Variables. First, we provide a set of variables for which we must find an assignment satisfying all the constraints in our problem. We declare 16 variables of domains $\{1, \ldots, 16\}$, thus already enforcing that all numbers must be between 1 and $n * n$. By doing so, the first two members of the constraint satisfaction problem $\mathcal{P} = \langle X, D, C \rangle$ are defined:

```
// 1. Initialization of variables
const int n = 4;
x = IntVarArray(*this, n*n, 1, n*n);
```

Figure 4.2 Variables for model in Fig. 4.1

Propagators. The last member of the CSP model corresponds to the constraints. In Gecode, implementation of constraints are called *propagators*. These objects are not created directly in the space's constructor; this responsibility is delegated to *post functions* that create the right propagators depending on parameters such as the arity or the consistency level of constraints. The constraints of our magic square problem are defined like this:

```
// 2. Initialization of propagators

// Matrix-wrapper for variables
Matrix<IntVarArray> m(x, n, n);

// We constrain variables already assigned
rel(*this, m(0,0) == 16);
rel(*this, m(1,1) == 10);
rel(*this, m(2,2) == 7);

// All numbers must be different
distinct(*this, x)

// Sum for each row/column/diagonal
const int S = n*(n^2+1)/2

// Rows and columns must be equal to S
for (int i = 0; i < n; i++) {
    linear(*this, m.row(i) == S);
    linear(*this, m.col(i) == S);
}

// Diagonals must also be equal to 34
linear(*this, m(0,0)+m(1,1)+m(2,2)+m(3,3) == S);
linear(*this, m(0,3)+m(1,2)+m(2,1)+m(3,0) == S);
```

Figure 4.3 Constraints for model in Fig. 4.1

4.1.3 Creation of the Search Strategy

The second component — the search strategy — is also defined inside the space constructor and created by post functions.

Branchers. The search strategy is represented by *branchers*. A brancher is responsible for giving a *branching choice* when the propagators are no longer able to do constraint propagation. The following function will create a brancher that will select the variable with

the smallest domain and branch on the smallest value.

```
// 3. Initialization of branchers
branch(*this, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

Figure 4.4 Brancher for model in Fig. 4.1

4.1.4 Solving the Problem

The previous components in the constructor are enough for describing our magic square. Aside from some small details, we now have a `Space` subclass for our problem named `MagicSquare`. Finding a solution is done by creating a `MagicSquare` object and passing it to a *search engine* such as depth-first search. As stated in Section 4.1.1, this corresponds to passing our object to the function `SOLVE` of Alg. 5. When doing so, the following will happen at each line:

2 Constraint propagation is done on the `MagicSquare` space; all instantiated propagators are going to filter variable domains according to their consistency levels until no more pruning can be done. At this point, the space will be at a *fixpoint* and will correspond to Fig. 1.5.

3-6 If any constraint is violated or any variable domain becomes empty while doing constraint propagation, the space will be invalid causing the function to return $\emptyset$ as no solution is possible. Otherwise, if all variables are fixed, we return the solution. Neither is the case in our example, so we go to line 7.

7 Here, we must make an hypothesis. Our brancher will give the branching choice $x_5 = 2$.

8 We split the space in two parts, like in Fig. 1.4. We will first call `SOLVE(`s_1`)` and redo the whole process given we have a new space modified by the branching choice.

Eventually, Gecode will find a solution using our model after 8 failures.

4.2 Basic Structure for Supporting Counting-Based Search in Gecode

The magic square can now be solved using the previous model. Our next step is to replace the brancher posted in Fig. 4.4 with a custom brancher using counting-based search.

Gecode is modular and extensible. A user is not limited to constructing a space that uses only available branchers and constraints in Gecode; it is possible to create custom branchers and propagators by inheriting from the base classes `Brancher` and `Propagator` respectively.

However, the abstractions provided were not sufficient to implement a brancher using counting-based search heuristics and this section describes the changes required to Gecode for this task. This was accomplished with one primary goal: make as few modifications to Gecode as possible. As we can see in Fig. 4.5, `MagicSquare` and our brancher can be created in the user model (a program that uses Gecode as a library). Our goal is to replace the function in Fig. 4.4 with a function posting a brancher of our own that uses counting-based search and Alg. 1 to make branching decisions.

Before doing so, we need a method for triggering solution density computation in propagators for generating $\langle propagator, variable, value, density \rangle$ tuples (i.e. a virtual method that we can overload to specify counting algorithms for different propagators).

Afterwards, even if we are able to call this method in a brancher and retrieve all tuples, we still won't be able make a CBS branching heuristic; making those heuristics require aggregating solution densities on propagators, variables and values. In Gecode, this is impossible as there's no unique id for variables and accessing them from different propagators can lead to different views of the same domain — an injective function may or may not be applied before access.

Addressing those concerns requires modifying multiple classes in the Gecode *kernel* and *Int* module, resulting in a small patch that can be divided in three parts, each of them described in the following subsections. After these modifications, we will be able to make our counting-based search branching heuristic.

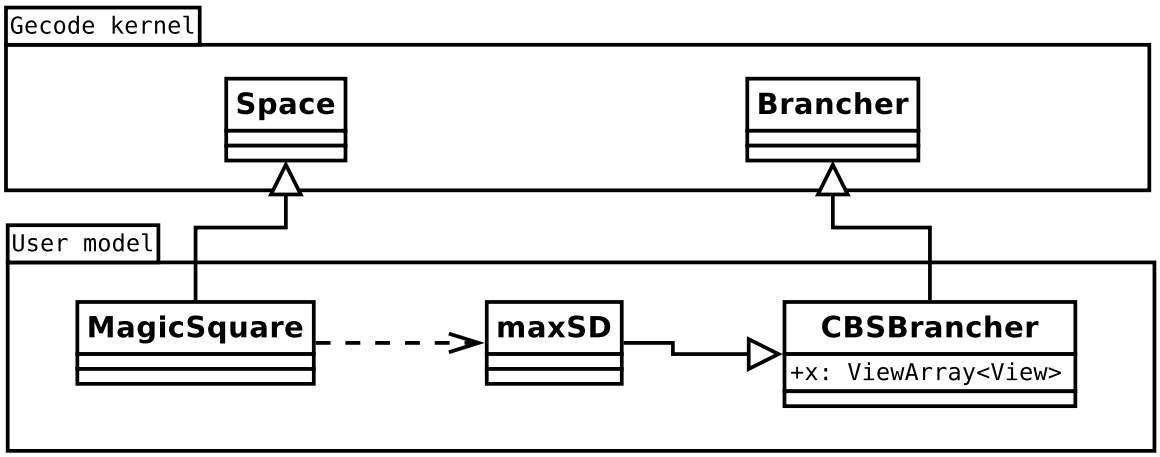


Figure 4.5 Modeling the magic square with a custom brancher using Gecode

Our goal — making as few modifications to Gecode as possible — may seem odd at first sight. The design we are presenting is actually our second iteration. In our first design, we did not pay attention to this factor and ended up with a patch of approximately 1500 lines. While it worked, its complexity made it very difficult to collaborate with the developers of Gecode. This is why we did a complete redesign, keeping only what is essential, to facilitate the integration of our patch.

4.2.1 Unique IDs for Variables

As implied in Alg. 1, we need a unique id for each constraint and variable in the model to compute a branching decision. Gecode did provide an id for propagators, but not for variables.

In Gecode, there are two possible abstractions over *variable implementations* (the object encoding the actual domains of variables): *variables* and *views*.

Variables are simply a read-only interface for variable implementations and they are used for modeling (see the protected member in Fig. 4.1). This makes sense because the model is declarative; we don't interact directly with domains.

On the other hand, views act as interfaces to variable implementations for propagators and branchers. They offer the ability to prune values from variable domains, and for this reason, must not be used for modeling.

For doing density aggregation, we need a unique id for variable implementation that is accessible from views. When creating a new variable implementation, its constructor has a

reference to the current space. Therefore, a global counter in `Space` that we increment in the constructor of `VarImp` can be used as a unique id. Accessing this new member amounts to adding a new method `id()` to the following classes:

- `VarImp`. Base-class for variable implementations.
- `VarImpView`. Base-class for variable implementation views. Gives a direct access to the domain.
- `DerivedView`. Base-class for derived views, for which an injective function may be applied before accessing the domain.
- `ConstView`. Base-class for constant views — views that mimic an assigned variable.

Those changes correspond to commit d4aa3 in https://github.com/Gecode/gecode.

4.2.2 Methods for Mapping Domains to their Original Values

Like previously said, propagators access variables via *views*. A view can remap every value of its domain. For example, accessing a variable with domain $D = \{1, 4, 8\}$ from a `MinusView` will yield $D' = \{-1, -4, -8\}$. Two propagators may use different views to the same variable, meaning we can't group densities by domain values.

To fix this, we added the following method to all views:

```
/// Return reverse transformation of value according to view
int baseval(int val) const;
```

Given that propagators already have unique ids, it is now possible to do aggregation of densities in `CBS_HEUR()` and Alg. 1 becomes:

```
1 SD = {}
2 foreach constraint c(x_1, ..., x_n) do
3     foreach unbound variable x_i ∈ {x_1, ..., x_n} do
4         foreach value d ∈ D_i do
5             SD[id(c)][id(x_i)][x_i.baseval(d)] = σ(x_i, d, c)
6 return CBS_HEUR(SD)                    ▷ Return a branching decision using SD
```

Algorithm 6: Modified version of Alg. 1 that uses unique ids for propagators and variables and take into account view transformations.

Those changes correspond to commit c39e3 in https://github.com/Gecode/gecode.

4.2.3 Computing Solution Distributions for Propagators

In Gecode, we can iterate over active propagators of the current space inside a brancher, but we do not have access to the internal state of propagators and their variables. To compute solution densities in propagators, a method must be added to the base class `Propagator`:

```
typedef std::function<void(unsigned int prop_id,
                           unsigned int var_id,
                           int val, double dens)> SendMarginal;
virtual void solndistrib(Space& home, SendMarginal send) const;
```

`SendMarginal` is a function that inserts solution densities in SD. Given we create a custom brancher with a method named `sendmarginal` and a private member named `SD`, making choices in our brancher will look like the following algorithm:

```
1 SD = {}
2 foreach constraint c do
3 |  c.solndistrib(space, sendmarginal)
4 return CBS_HEUR(SD)                    ▷ Return a branching decision using SD
```

Algorithm 7: Modified version of Alg. 6 closer to how Gecode interacts with propagators.

Those changes correspond to commit 6957b in https://github.com/Gecode/gecode. With these three commits, it is possible to create a counting-based search brancher given we implement specialized counting algorithms for constraints by overloading `solndistrib` in the correct propagators.

4.2.4 Accessing Variable Cardinalities Sum in Propagators

Lastly, a final method is required in `Propagator` if we want to make counting-based search efficient.

If all variable domains in a propagator stay identical after branching and propagating, we can reuse previous calculated densities when making the next branching choice. However, as shown in Alg. 7, we recompute every solution distributions from scratch each time we make a choice. We can fix this by adding the following method in `Propagator`:

```
typedef std::function<bool(unsigned int var_id)> InDecision;
virtual void domainsizesum(InDecision in, unsigned int& size,
                           unsigned int& size_b) const;
```

As we can only remove values from domains during propagation, a variable with equal cardinality before and after propagation has the same domain. This logic holds true for an entire propagator: if the sum of its variable domain cardinalities is equal before and after propagation, all the variables in the propagator are unchanged, meaning we don't have to recompute densities. Those changes also correspond to commit 6957b in https://github.com/Gecode/gecode.

This method will be used in our brancher to avoid unnecessary recomputation of solution densities.

4.3 Counting Algorithms

Section 4.2 gave the basic structure for creating a custom brancher that uses counting-based search. However, we still need counting algorithms for propagators created when specifying a model to get solution densities and make branching choices.

Let's take the example of our Magic Square Problem again. When instantiating the model of Fig. 4.1, the following propagators are created (using their default consistency):

- A single `Int::Distinct::Val` propagator for the `distinct` constraint.
- Several `Int::Linear::Bnd` propagators for the `linear` constraints.

Thus, we need to overload `Int::Distinct::Val::solndistrib` and `Int::Linear::Bnd::solndistrib` to specify counting algorithms (alongside `domainsizesum` for both classes). Without this, our brancher will not be able to give branching choices; the more propagators we support, the more problems we can solve.

In this thesis, we implemented counting algorithms for `distinct`, `linear`, and `extensional` (regular expression) constraints [Pesant and Quimper (2008)][Zanarini and Pesant (2010)]. Annex A gives implementation details about each counting algorithm alongside a generic template for implementing them.

4.4 Creating a Brancher Using Counting-Based Search

We now have everything for creating our custom brancher as shown in Fig. 4.6.

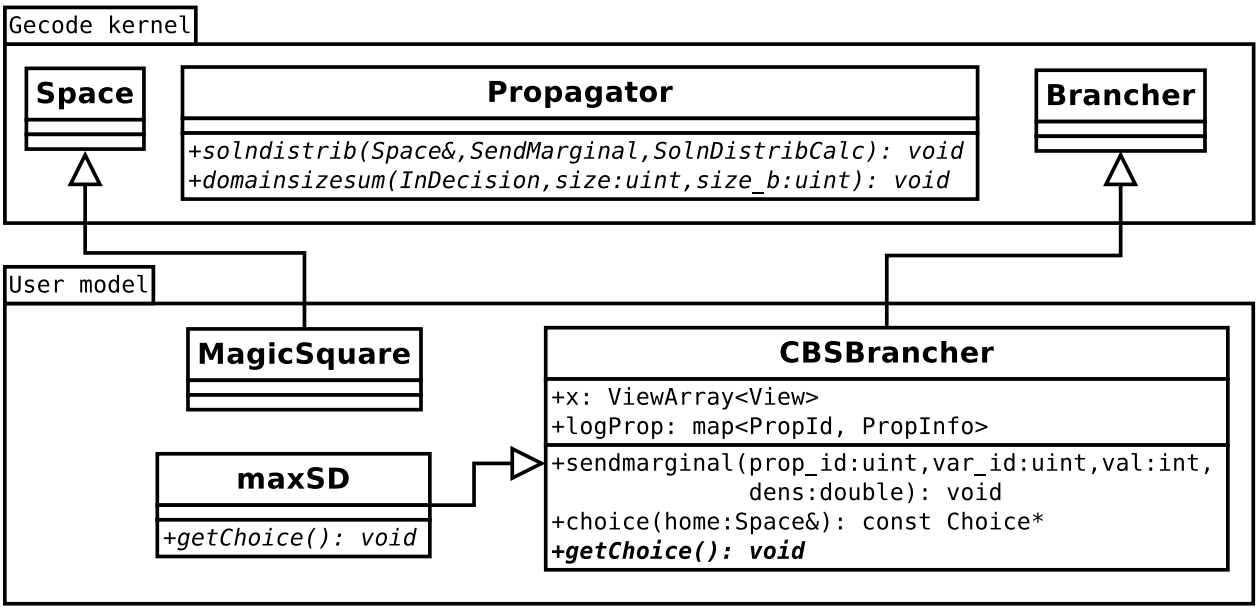


Figure 4.6 Principal components in relation with `CBSBrancher`

The logic for making branching choices — described in Alg. 7 — is located in the method `choice`. This method stores solution densities in a map called `logProp` and uses `domainsizesum` to evaluate if densities for a given propagator need to be recalculated between propagations.

As counting-based search is a family of branching heuristics, the concrete brancher is created by inheriting from `CBSBrancher` and overloading `getChoice` for specifying how we make choices from all $\langle propagator, variable, value, density \rangle$ tuples. In this case, we use the branching heuristic *maxSD*, but other heuristics are possible such as *aAvgSD*, *maxRelSD*, etc. (as explained in [Zanarini and Pesant (2007)]).

In Section 5.2, we benchmark *maxSD* against other generic branching heuristics for three different problems.

CHAPTER 5 EMPIRICAL EVALUATION

In this chapter, an empirical evaluation of all work discussed in Chapter 3 and Chapter 4 is given. Section 5.1 evaluates the improvements to the `alldifferent` and `spanningTree` constraints. Section 5.1.3 evaluates the generic method proposed in Section 3.3 to avoid calling counting algorithms at all nodes. Finally, Section 5.2 compares our implementation of CBS with other generic branching heuristics in Gecode.

All graphs in this chapter are interpreted the same way across all benchmarks, with each benchmark accompanied by two graphs. The first one gives the percentage of instances solved with respect to the number of failures — if we have 100 instances of a particular problem and a cutoff of 1000 failures when backtracking, $f(1000) = 40$ means that we are able to solve 40 instances. The second graph is similar except time is used as the cutoff.

5.1 Impact of our Contributions

5.1.1 Acceleration of `alldifferent`

The Quasigroup Completion Problem consists in filling a $m \times m$ grid with numbers such that each row and column contains every number from 1 to m (#67 in CSPLib [Pesant (2018)]). It is very similar to Sudokus with the only difference being that the grid is not separated in sub-grids where all numbers must be different. Surprisingly, this makes the problem a lot harder.

This problem can be described using an `alldifferent` constraint on each row and column, making it ideal for testing our improvements to its counting algorithms. Gecode's distribution already includes a model with branching heuristics *afc* (weighted degree) and *size* (smallest domain), both with lexicographic value selection. The focus of this experiment is to compare the performance of *maxSD* using Algorithm 2, 3 and 4, but we also tried the other generic heuristics for comparison.

We use 20 instances of sizes 90 to 110 with 25% of entries filled, generated as in [Gomes and Shmoys (2002)]. That ratio of filled entries may not yield the hardest instances for that size but our goal here is to have a lot of shared values between variables in order to emphasize the improvement of Algorithm 3 over 2.

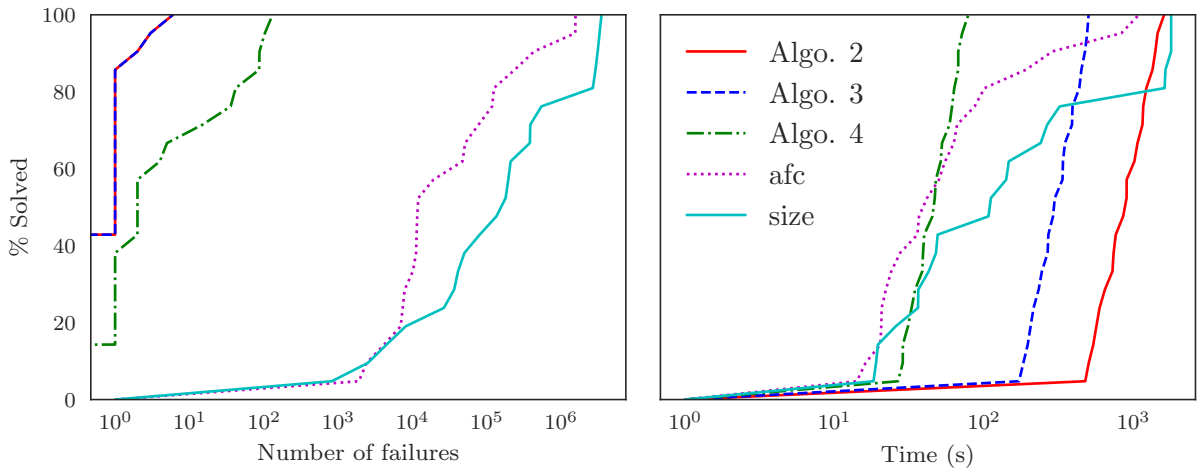


Figure 5.1 Percentage of Quasigroup Completion instances solved w.r.t. time and number of failures

First we observe that *maxSD* guides search more effectively by solving all instances in several orders of magnitude fewer failures than *afc* and *size*. As expected, because of the lack of normalization, Algorithm 4 is less accurate than the other two (which share the same number of failures), but still about one order of magnitude faster.

Even with around 3 orders of magnitude fewer failures, *afc* is faster than Algorithm 4 until around 60% of instances solved. This is because making a branching choice using *afc* is near-instantaneous when compared to *maxSD*. Before our improvements, *maxSD* was slower than both *afc* and *size*.

5.1.2 Acceleration of `spanningTree`

To test improvements introduced in Section 3.2.1 and 3.2.2, we designed a simple model to find Hamiltonian paths with Gecode's `path` constraint and a redundant `spanningTree` constraint that computes solution densities according to our `spanningTree` counting algorithm. The `spanningTree` constraint is expressed on binary edge variables as opposed to vertex successor variables for the `path` constraint — the two variable representations are channeled.

In Fig. 5.2 improvements to the `spanningTree` counting algorithm are tested on this model with *maxSD* branching on binary edge variables — for comparison we also tried heuristics *afc* and *size* branching on vertex successor variables (and trying values in lexicographic order). We use 30 graphs over 60 to 234 vertices taken from the FHCP Challenge Set [Haythorpe (2015)].

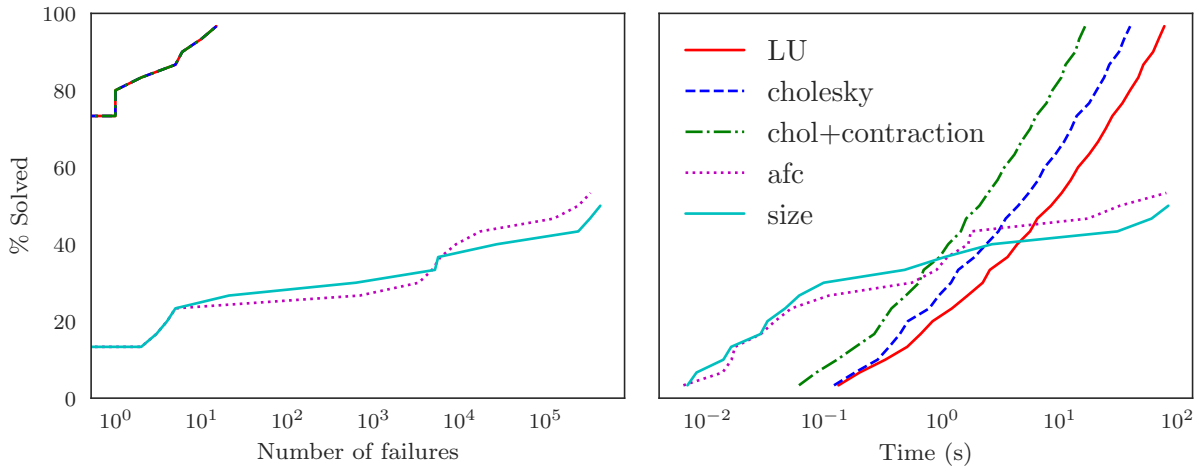


Figure 5.2 Percentage of Hamiltonian Path instances solved w.r.t. time and number of failures with a 5 minute cutoff

Again we observe that *maxSD* (with the curves of its three algorithms coinciding in the first graph) guides search much more effectively by solving the instances in several orders of magnitude fewer failures than *afc* and *size*. At one second of computation time, all three heuristics solve about the same number; at ten seconds, *maxSD* solves almost all of them whereas *afc* and *size* solve about half.

5.1.3 Avoiding Systematic Recomputation

In this section, we test the generic method that we proposed for avoiding systematic recomputation of solution densities in Section 3.3, first on the Quasigroup Completion Problem with Algorithm 4 (Fig. 5.3), and then on the Hamiltonian Path Problem with Algorithm *chol+contraction* (Fig. 5.4), using the same model and instances as in Section 5.1.1 and 5.1.2. The variable ρ represents the recomputation ratio, with $\rho = 1$ meaning we always recompute.

While $\rho = 0.95$ is indeed faster than full recomputation in Fig. 5.3, a lot of accuracy is lost for the small speed advantage we gain. It's possible this tradeoff may not be worth it for harder instances. It is also interesting to see that $\rho = 0.8$ performs about the same as $\rho = 0.95$.

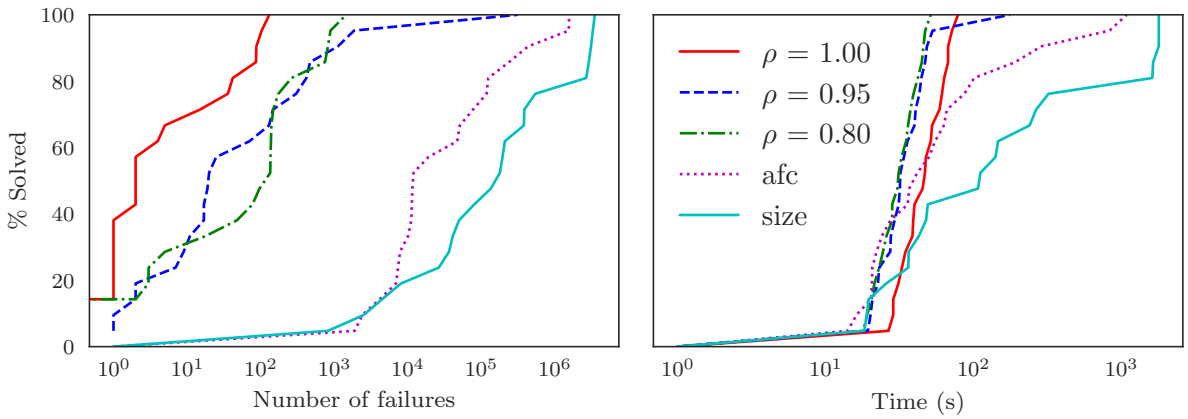


Figure 5.3 The effect of different recomputation ratios for the Quasigroup Completion Problem, using the same instances as in Section 5.1.1

However, as shown in Fig. 5.4, the acceleration brought by a recomputation ratio of 0.8 for the `spanningTree` constraint is more pronounced than both of our improvements for its counting algorithms. Indeed, computing solution densities for this constraint is very expensive; the tradeoff between precision and speed is worthwhile for this problem.

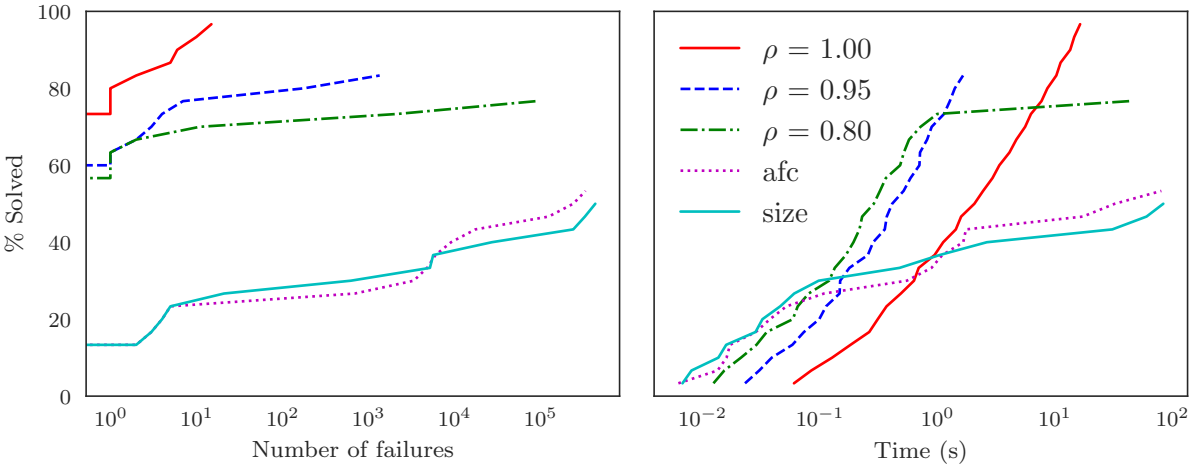


Figure 5.4 The effect of different recomputation ratios for Hamiltonian Path Problem, using same instances as in Section 5.1.2

5.2 Benchmarking our Implementation in Gecode

In this section, we show additional benchmarks comparing *maxSD* to other generic heuristics. As Gecode already includes a set of models in its distribution alongside a choice of branching

heuristics for each model, those benchmarks are easy to make. Using *maxSD* in a model is as simple as adding an `#include` for our brancher and calling its post function `cbsbranch`.

Moreover, these benchmarks support our efforts in showing that counting-based search is a worthwhile addition to Gecode. All presented models do not use our technique for avoiding systematic recomputation.

5.2.1 Magic Square

Gecode has a model for the Magic Square Problem but doesn't have partial instances, so we created a set of 40 instances of size 9x9. Gecode provides two branching strategies for this model:

- **size**. Choose the variable x_i with the smallest domain and branch on $x_i \leq mean(D_i)$. This is a direct application of the fail-first principle seen in Section 2.1.1 with the domain size as the constraining criteria.
- **afc-size**. Select variable x_i with largest accumulated failure count (AFC) divided by domain size and branch on $x_i \leq mean(D_i)$. AFC is an adapation from WDEG (Section 2.1.3).

Those two heuristics were tested against *maxSD* in Fig. 5.5 using Algorithm 3 for the `alldifferent` constraint. At around 1000 failures *maxSD* solves more instances than *afc-size* and *size*. However, even if *maxSD* is always equal or better in terms of failures, it solves fewer instances below 1 second. Our extra accuracy comes at a cost: we take additional time to call our counting algorithms when branching. This can lower greatly the node/sec explored when compared with a branching heuristic like *size*, where a single pass on all variables is enough to make a choice.

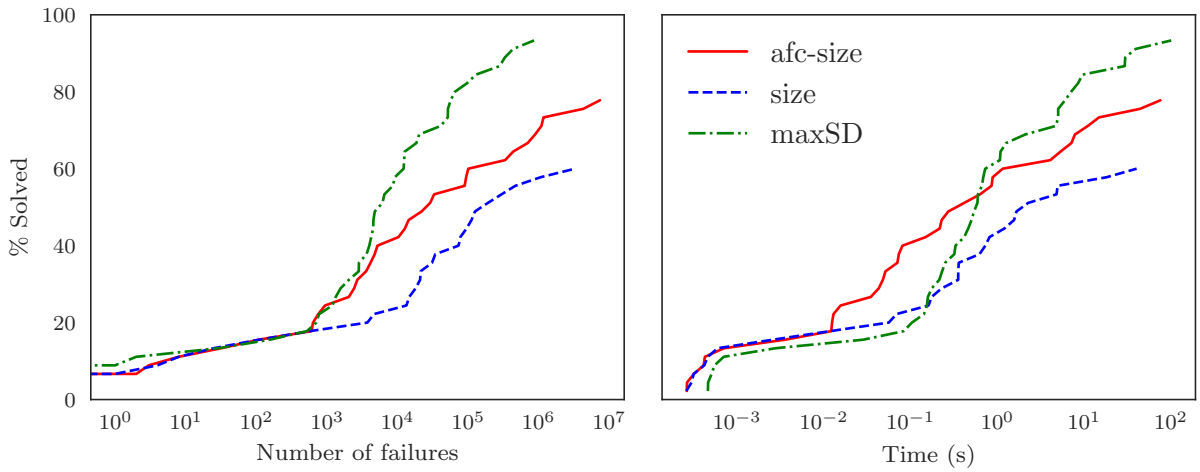


Figure 5.5 Percentage of magic square instances solved w.r.t. time and number of failures

5.2.2 Quasigroup Completion with Holes Problem

Gecode has a set of 75 partial Latin Square instances with sizes ranging from 10x10 to 70x70. Again, Gecode's model gives the choice of two branching strategies. They are identical to the two previous branching heuristics in the magic square model except for the choice value; they branch on $x_i = min(D_i)$ instead of $x_i \leq mean(D_i)$. Even if this model was already benchmarked in Section 5.1.1, it shows how *maxSD* performs on Gecode's instances; our previous instances were not created to be hard, they were created to have a lot of holes. We also use Algorithm 4 for the `alldifferent` constraints.

We can see in Fig. 5.6 that our branching strategy greatly outperform the two other alternatives, both in the number of failures and solving time.

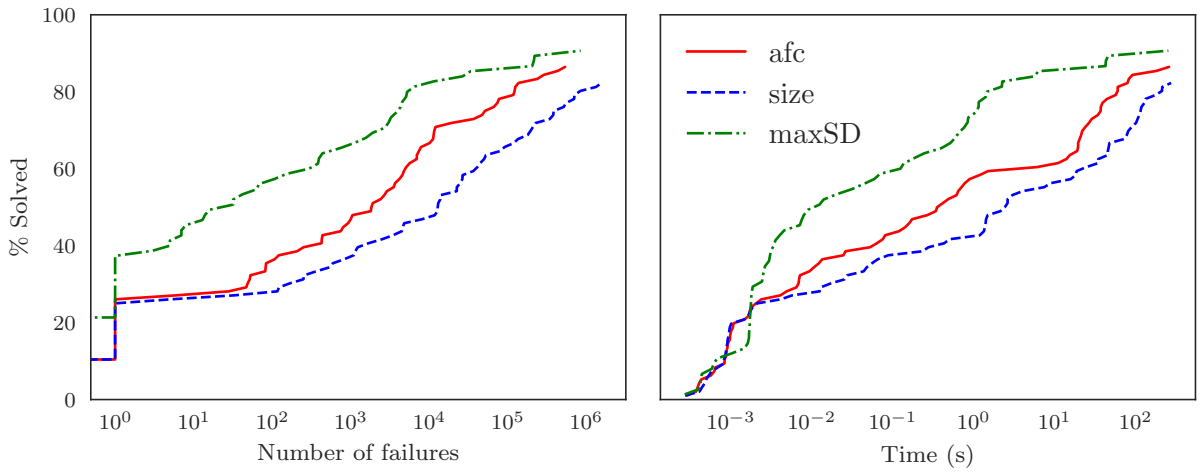


Figure 5.6 Percentage of latin square instances solved w.r.t. time and number of failures

5.2.3 Langford Number Problem

The Langford Number is a problem where we have $2n$ numbers from 1 to n, with two copies of each number:

$$\{1, 1, 2, 2, \ldots, i, i, \ldots, n, n\} \tag{5.1}$$

The goal is to find a sequence where each pair of numbers i are $i + 1$ positions apart. The sequence $\{4, 1, 3, 1, 2, 4, 3, 2\}$ is a solution for $n = 4$.

This problem can be modeled using only `regular` constraints, for which we have a counting algorithm. Fig. 5.7 compares *maxSD* with two other default branching strategies provided with Gecode. The *afc* strategy selects the variable in the same manner as previously discussed, while the *none* strategy selects the first unassigned variable. Both strategies branch on $x_i \leq mean(D_i)$.

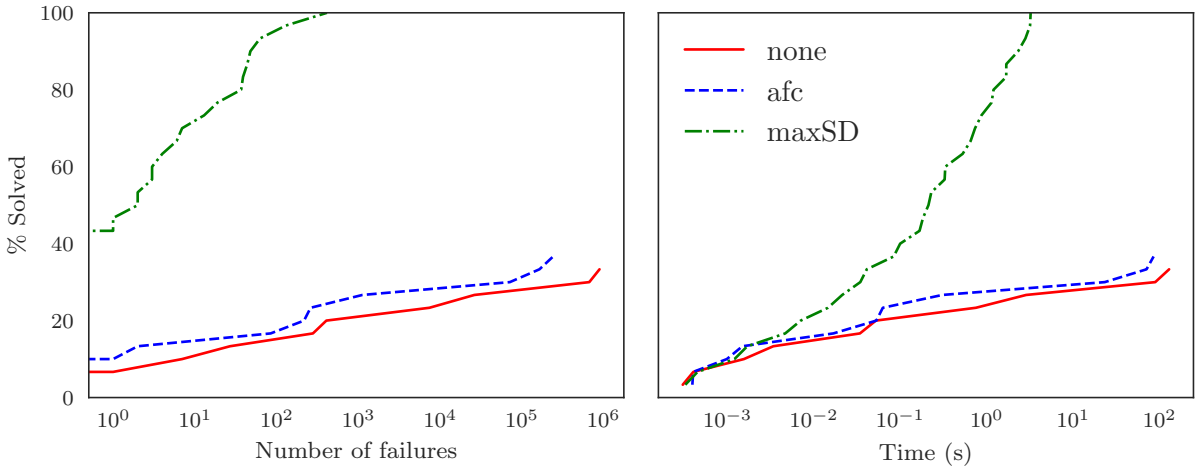


Figure 5.7 Percentage of langford instances solved w.r.t. time and number of failures

It is interesting to see that *afc* and *none* gives almost the same results, as opposed to *maxSD* which directs the search much more effectively.

CHAPTER 6 BRANCHING HEURISTIC CREATION USING A LOGISTIC REGRESSION

Counting-based search relies on specialized counting algorithms to compute solution densities in each constraint; this information can be used in many ways to design branching heuristics. Choosing the right one among all possible choices requires experimentation when solving a new problem. Even then, the best performing heuristic may vary between different instances.

In Section 2.3, we briefly reviewed different studies tackling this problem with *hyper-heuristics* or adaptations of older methods like *FORR*. However, this chapter is better described as exploratory; we did not try to improve the state of the art for learning or selecting heuristics. Rather, we wanted to see how far we could go by using a simple method before looking at more complex algorithms; simpler methods are easier to reason about, and in this sense may provide more insights than their complex alternatives.

6.1 Problem Definition

Normally a branching heuristic will select a variable according to various information. The next list gives some examples:

- Variable cardinality
- Degree or number of constraints on the variable
- Weighted-degree (Section 2.1.3)
- Activity (Section 2.1.4)
- Variable domain

Afterwards, the branching heuristic will modify the domain of the variable by splitting it or selecting a single value. Such a process is in two steps: variable selection, and value selection. In this regard, counting-based search branching heuristics are different; they select both the variable and the value at the same time. This is due to the fact that a density is computed for every variable-value pair.

This density is an additional feature for making a branching choice. Each CBS branching heuristic is characterized by the way it handles this new information. In addition to examples already introduced in Section 2.1.5 — *maxSD* and *aAvgSD* — we also have: *maxRelSD*, which

selects the maximum of the solution densities after subtracting the average density in the corresponding variable; or *maxRelRatio*, which divides by the average density of the variable. Both of these examples are taken from [Pesant et al. (2012)], but there are many more in [Zanarini (2010)].

Each of these branching heuristics will give a different *score* to each variable-value pair, and will branch on the variable-value with the highest score in a given node. Therefore, when solving a satisfaction problem, a heuristic that gives high scores to variable-value pairs that belong to solutions is going to find a solution faster; the score of a branching heuristic can be seen as predicting if a variable-value pair will segment the solution space towards a solution.

We can take this a step further and classify each variable-value pair as good —it leads to a solution— or bad —it doesn't lead to a solution. However, this is hard if there are multiple solutions for an instance; a particular choice may be part of a solution in a given sub-tree but not in another one. This problem can be lifted by only considering instances of a problem for which the solution is unique. While this is restricting, this allows us to gather data in an offline database like the following in table:

Table 6.1 Offline database with densities for multiple single solution instances of the same problem.

					dens	maxSD	aAvgSD	...	in_sol
exec_id	node_id	prop_id	var_idx	val				...	
3	12	6	3	2	0.3	0.6	0.45	...	0
				3	0.4	0.4	0.4	...	1
				8	0.3	0.5	0.48	...	0
			8	2	0.8	0.8	0.5	...	1
				9	0.2	0.4	0.22	...	0

For a given problem with unique solutions, we can gather the score of all CBS branching heuristics, and label each variable-value, in each propagator, in each *satisfiable* node, as a good or bad branching choice for a given instance. We only gather nodes that lead to a solution in the search tree (satisfiable nodes), because we don't want to learn from incoherent states.

This gives us a classification problem: given the score of all heuristics, predict if an assignment is in the final solution or not. For our purpose, the logistic regression was a good choice: it is simple and easy to understand, all our attributes are numerical, and it is fast to train. Moreover, it is also a starting point for more complex techniques.

6.2 Experiment

In this section, two experiments on the same training and testing set are described. As shown in Figure 6.1, we train a logistic function and use it as a new branching heuristic in both experiments. Each experiment differs in the subset of features — CBS heuristics — it uses for the logistic regression. But why not choosing all 9 features at once?

Figure B.1 shows, in the training set, how the scores of each branching heuristic separate values leading to a solution from those who aren't. As we can see, many features are redundant: their distribution gives the same separation. A logistic regression may not be optimal if such features are used together.

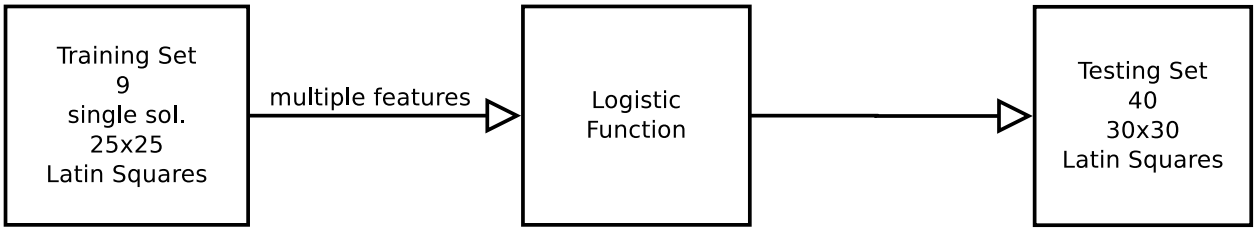


Figure 6.1 Logistic function as a new branching heuristic

6.2.1 First Logistic Function

For the first experiment, we chose two different features — h_2: *aAvgSD* and h_5: *wSCAvg* — that seemed to be redundant according to Figure B.1. We wanted to see how the logistic function would perform on unseen instances in the testing set. The logistic regression gave the following function:

$$(1 + exp(-(9.2h_2 - 2.4h_5 - 2.9))^{-1} \tag{6.1}$$

When used as a new branching heuristic in the testing set, this logistic function is worse than both of its branching heuristic taken separately:

Table 6.2 Logistic function combining $h_2 \wedge h_5$ against both branching heuristic.

heur	avg. failures for solved instances	nb. instances solved w/ 100 000 failures cutoff
h_2	22945	29
h_5	26230	30
logReg	26284	23

This experiment supports the claim that combining redundant features is not a good idea.

6.2.2 Second Logistic Function

In the second experiment, we chose to combine two branching heuristics with different distributions according to Figure B.1. When using h_2: *aAvgSD* and h_3: *maxRelSD*, we obtain the following logistic function:

$$(1 + exp(-(4.7h_2 + 6.5h_3 - 2.6))^{-1} \tag{6.2}$$

This times, it outperforms each of its individual heuristics:

Table 6.3 Logistic function combining $h_2 \wedge h_3$ against both branching heuristic.

heur	avg. failures for solved instances	nb. instances solved w/ 100 000 failures cutoff
h_2	22945	29
h_3	19298	26
logReg	17156	34

However, one could ask if we simply got lucky. In other words, maybe a slight change to this logistic function is enough to make it perform worse than its heuristics. To find out, we did another experiment that we describe in the next paragraphs.

A logistic regression returns a linear function with specific weights that combine every feature. It can then be used inside a logistic function to give scores between 0 and 1. In our case, we learned to following linear function:

$$t(h_2, h_5) = 4.7h_2 + 6.5h_3 - 2.6 \tag{6.3}$$

This equation corresponds to a plane in 3 dimensions. If we want to change its coefficients slightly, we can simply rotate the plane around the mean value of both heuristics, like shown in Figure 6.2. If we rotate t clockwise, we will eventually get a plane where $h_3 = 0$, meaning it will perform identically to h_2 when put inside the logistic function. We can get the same result for h_3 if we rotate counterclockwise.

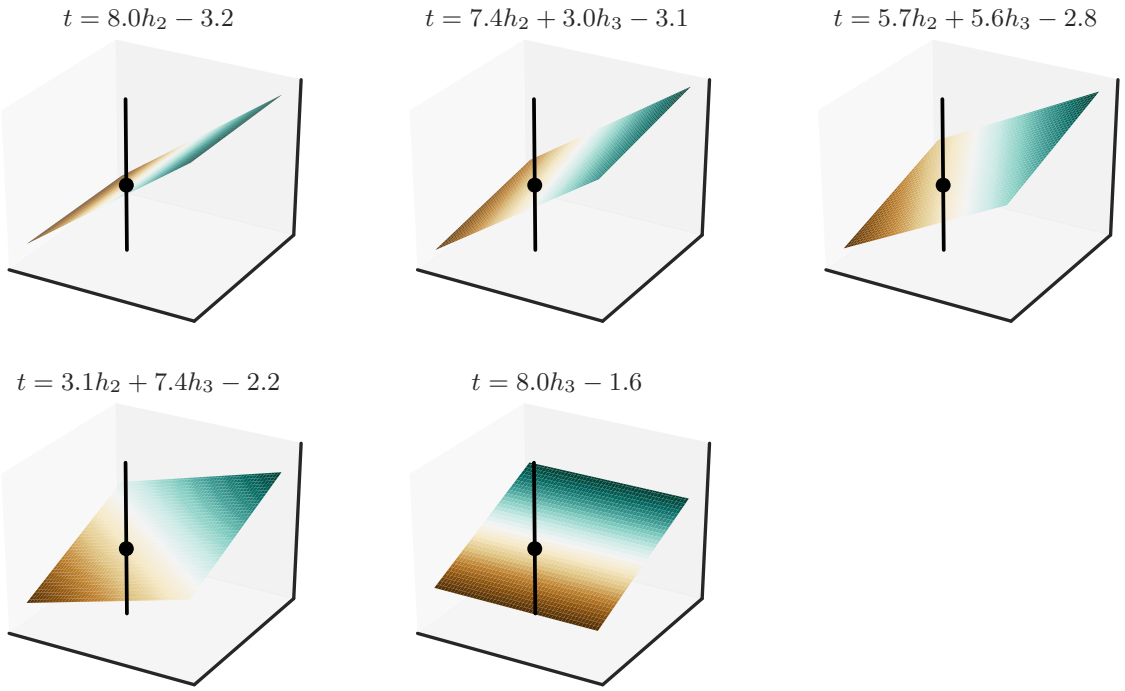


Figure 6.2 Rotation of $t(h_2, h_3)$ around the mean value of h_2 and h_3

Between those two extremes, there are 90 degrees, each corresponding to a different logistic function. Furthermore, according to Murphy's law, if you give a student access to a computer cluster, he will eventually do something stupid. Jokes aside, Figure 6.3 plots how many instances of our test set were solved under 100 000 failures for each of those 90 logistic functions.

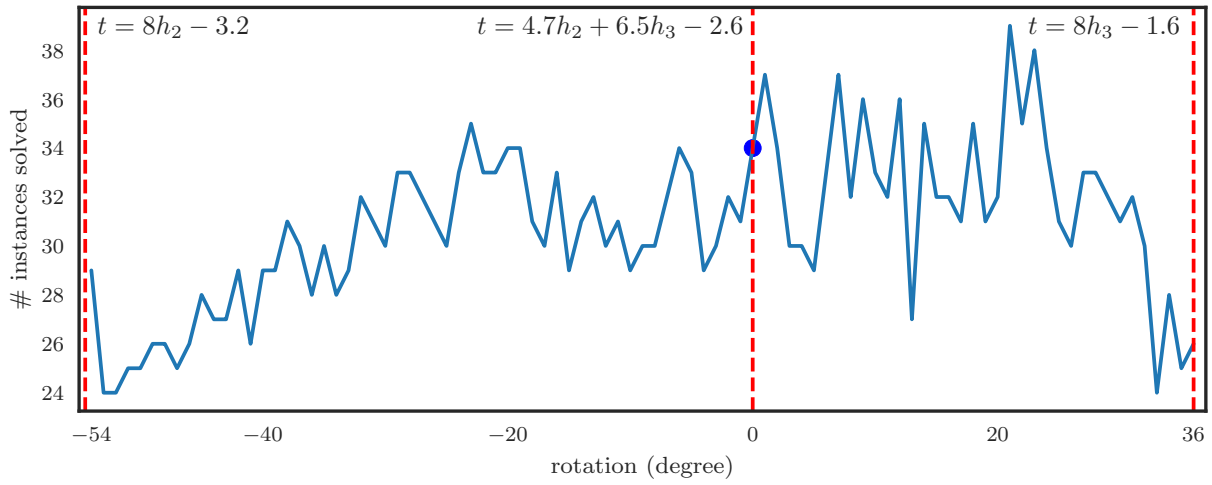


Figure 6.3 Landscape of performance for 90 logistic functions

We can see that combining h_2 and h_3 generally yields better performing heuristics than both h_2 (far left) or h_3 (far right) alone. Also, our logistic regression found a linear combination that was among the best ones.

CHAPTER 7 CONCLUSION

This thesis was concerned with counting-based search, a paradigm for designing branching heuristics in constraint programming. The main narrative underlying this work can be stated as follows: bring CBS to a new constraint programming solver and showcase it as a solid choice when considering other generic alternatives.

In the next sections, I will summarize our work while giving a more personal narrative. I think a summary where I give the driving events and motivations behind each effort is more interesting and meaningful; it breaks the false perception that everything was all figured out from the beginning.

7.1 Discussion of Contributions

In Chapter 4, a minimal patch for supporting counting-based search in Gecode is presented. Finding a way to make CBS work in Gecode was our first goal in this Master's degree; all work done in other chapters depends on this implementation. It would have been possible to use our existing framework in ILOG Solver. However, as CBS is not officially included in ILOG solver, this would have been detrimental to one of our main objectives: make CBS more accessible.

We chose an open-source constraint programming solver — Gecode — because it would be easier to test ideas without restrictions. This has proven a good choice because the small design presented in this thesis is the aftermath of a lot of experimentation and competing ideas; getting a working implementation was a first step towards a clean one. We think our efforts in making this patch small is one of the main reasons — alongside our experimental results — that CBS was accepted in the solver. As a result, our implementation will be available in the next release of Gecode for practitioners.

Although Chapter 3 is introduced in this thesis before our Gecode implementation, the work described in this chapter came after introducing CBS in Gecode. With `alldifferent` the first constraint for which we had a working counting algorithm, our first benchmarks were on the Quasigroup Completion with Holes Problem. We quickly realized that, even if CBS led to solutions with fewer backtracks, it was still an order of magnitude slower than other generic alternatives because of all the extra work we do in computing solution densities. The algorithms proposed in Section 3.1 are the result of all our efforts in making *maxSD* faster than generic alternatives for Gecode's QCP instances in Fig. 5.6. Altough our main motivation

was the QCP problem, `alldifferent` is a central constraint in CP, and accelerating its counting algorithms leads to CBS performing better in every model that uses it.

Next, the idea of improving the `spanningTree` counting algorithm in Section 3.2 is the result of a competition we gave in the undergraduate course *INF4705* for finding Hamiltonian paths. We had a very hard instance of 558 nodes and 837 vertices from the same dataset evaluated in Section 3.2. Out of 50 teams, only 3 were able to find a valid solution for this instance under 3 minutes. Out of curiosity, we tried a CP model using CBS, and to our surprise, it was able to find a solution with 2 failures. However, this solution was found in 25 minutes. Thus, with the goal of breaking 3 minutes, we were able to go as low as 20 seconds for this instance — inverting matrices of size 558x558 during the whole search wasn't very effective.

Lastly, the experiment in Chapter 6 came from a desire to try out new techniques in machine learning. However, we quickly realized that complex methods like neural networks were less interesting as they don't offer a lot of *insights* about what has been learned in opposition to simpler methods like decision trees or logistic regressions; with simple methods, we can look at the learned models and understand various things. While limited, we indeed learned a new heuristic that is outperforming its *low-level* heuristics.

7.2 Limits and Constraints

It would be strange to say that our improvements of counting algorithms in Section 3 have *limitations*. We proposed better algorithms in both theory and practice; they have better asymptotic complexities and are faster in our benchmarks when choosing hard instances. One could argue that our approach for avoiding systematic recomputation has the limitation that it has lower precision, but we see it more as a trade off between speed and precision. In my opinion, *limitations* arise when considering practical matters, which comes from the choice of a paradigm — counting-based search — and its actual implementation in Gecode.

While generic and powerful, using counting-based search indeed limits the constraints we can use for modeling in practice. In other words, the problems we can solve are limited by the constraints we support (using CBS with some uninstrumented constraints is possible, but it is less effective because we miss solution densities). Furthermore, supporting a new constraint is difficult: we have to design a new counting algorithm and provide an efficient implementation. As CP solvers support a lot of different constraints, designing counting algorithms is a long-term project. This is a price we must pay for having a family of generic branching heuristics that performs well and adapts to the problem formulation.

Next, the fact that our patch is small also comes at a cost. In Section 4.4, great emphasis is

put on Fig. 4.6 and the fact that our brancher can be created outside of Gecode, thus making our patch smaller. While anyone can use our work in Gecode to make a CBS brancher, making an efficient one is difficult in practice. Until we propose another patch for adding our brancher in Gecode, we will have to distribute it by separate means to our potential users.

Finally, like we said earlier, our experiment in Chapter 6 is rather limited. We only tried one problem — Quasigroup Completion — and trained on instances with unique solutions; we are not yet sure how this would transfer to problems with multiple solutions. Moreover, accumulating all these densities in an offline database and keeping reasonable performance is not trivial.

7.3 Future Research

Hopefully, most of the issues mentioned in the previous section can be fixed. Even if designing counting algorithms is hard, each new one improves the scope of counting-based search. Moreover, our design makes it easy to add them to Gecode. Afterwards, submitting a patch for including our brancher in Gecode is also in our projects.

Of course, the work in this thesis can be improved in several ways. Aside from the obvious suggestion of further improving counting algorithms, and in the same spirit as our proposed technique for avoiding systematic recomputation of solution densities in Section 3.3, the amount of work we do could be more *adaptive*. As triggering counting algorithms is expensive, finding clever ways to avoid calling them with minimal impact on the precision of the search is a worthwhile study path. Here are some ideas:

- If we knew which constraints are the most susceptible of returning variable-value pairs with high densities, we could avoid computation when using a branching heuristic such as *maxSD*.
- In Section 3.3, we use the sum of all variable domain cardinalities as a metric for judging if we can reuse densities from the previous node. Other metric could be investigated.

Besides, we could also benefit from using techniques for intelligently adapting consistency levels of propagators (as discussed in Section 2.2); weaker consistency levels also mean faster counting algorithms in some cases.

Finally, adaptive branching heuristics, analog to the experiment discussed in Section 6, could further improve the performance of CBS.

REFERENCES

R. Barták, "On-line guide to constraint programming," April 2018. [Online]. Available: http://ktiml.mff.cuni.cz/~bartak/constraints/index.html

F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proceedings of the 16th European Conference on Artificial Intelligence*, ser. ECAI'04. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2004, pp. 146–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=3000001.3000033

S. Brockbank, G. Pesant, and L. Rousseau, "Counting spanning trees to guide search in constrained spanning tree problems," in *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, ser. Lecture Notes in Computer Science, C. Schulte, Ed., vol. 8124. Springer, 2013, pp. 175–183.

E. K. Burke, M. Gendreau, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *JORS*, vol. 64, no. 12, pp. 1695–1724, 2013. [Online]. Available: https://doi.org/10.1057/jors.2013.71

S. Chaiken and D. J. Kleitman, "Matrix tree theorems," *Journal of Combinatorial Theory, Series A*, vol. 24, no. 3, pp. 377 – 381, 1978.

J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the scalapack lu, qr, and cholesky factorization routines," *Sci. Program.*, vol. 5, no. 3, pp. 173–184, Aug. 1996.

A. Delaite and G. Pesant, "Counting weighted spanning trees to solve constrained minimum spanning tree problems," in *CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, ser. Lecture Notes in Computer Science, D. Salvagnin and M. Lombardi, Eds., vol. 10335. Springer, 2017, pp. 176–184.

S. L. Epstein, "For the right reasons: The FORR architecture for learning in a skill domain," *Cognitive Science*, vol. 18, no. 3, pp. 479–511, 1994. [Online]. Available: https://doi.org/10.1207/s15516709cog1803_4

S. L. Epstein, E. C. Freuder, R. J. Wallace, A. Morozov, and B. Samuels, "The adaptive constraint engine," in *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*,

ser. Lecture Notes in Computer Science, P. V. Hentenryck, Ed., vol. 2470. Springer, 2002, pp. 525–542. [Online]. Available: https://doi.org/10.1007/3-540-46135-3_35

S. Gagnon, "Gecode extension for counting-based search," https://github.com/SaGagnon/gecode-5-extension, 2017.

Gecode Team, "Gecode: Generic constraint development environment," 2017, available from `http://www.gecode.org`.

C. Gomes and D. Shmoys, "Completing quasigroups or latin squares: A structured graph coloring problem," in *Computational Symposium on Graph Coloring and Generalizations*, 01 2002.

R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artif. Intell.*, vol. 14, no. 3, pp. 263–313, 1980. [Online]. Available: https://doi.org/10.1016/0004-3702(80)90051-X

M. Haythorpe, "Fhcp challenge set," 2015. [Online]. Available: http://fhcp.edu.au/fhcpcs

IBM, "Ilog is now part of ibm," March 2018. [Online]. Available: http://www-01.ibm.com/software/info/ilog/

L. Michel and P. 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*, N. Beldiceanu, N. Jussien, and É. Pinson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 228–243.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 530–535. [Online]. Available: http://doi.acm.org/10.1145/378239.379017

J. C. Ortiz-Bayliss, E. Özcan, A. J. Parkes, and H. Terashima-Marín, "Mapping the performance of heuristics for constraint satisfaction," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*. IEEE, 2010, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CEC.2010.5585965

J. C. Ortiz-Bayliss, H. Terashima-Marín, S. E. Conant-Pablos, E. Özcan, and A. J. Parkes, "Improving the performance of vector hyper-heuristics through local search," in *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA,*

July 7-11, 2012, T. Soule and J. H. Moore, Eds. ACM, 2012, pp. 1269–1276. [Online]. Available: http://doi.acm.org/10.1145/2330163.2330339

A. Paparrizou and K. Stergiou, "Evaluating simple fully automated heuristics for adaptive constraint propagation," in *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*. IEEE Computer Society, 2012, pp. 880–885. [Online]. Available: https://doi.org/10.1109/ICTAI.2012.123

G. Pesant, "Achieving domain consistency and counting solutions for dispersion constraints," *INFORMS Journal on Computing*, vol. 27, no. 4, pp. 690–703, 2015. [Online]. Available: https://doi.org/10.1287/ijoc.2015.0654

——, "Counting solutions of csps: A structural approach," in *In IJCAI-05*, 2005, p. 260.

——, "Quasigroup completion," April 2018. [Online]. Available: http://csplib.org/Problems/prob067/

G. Pesant and C. Quimper, "Counting solutions of knapsack constraints," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, Paris, France, May 20-23, 2008, Proceedings*, ser. Lecture Notes in Computer Science, L. Perron and M. A. Trick, Eds., vol. 5015. Springer, 2008, pp. 203–217. [Online]. Available: https://doi.org/10.1007/978-3-540-68155-7_17

G. Pesant and A. Zanarini, "Recovering indirect solution densities for counting-based branching heuristics," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, ser. Lecture Notes in Computer Science, T. Achterberg and J. C. Beck, Eds., vol. 6697. Springer, 2011, pp. 170–175. [Online]. Available: https://doi.org/10.1007/978-3-642-21311-3_16

G. Pesant, C.-G. Quimper, and A. Zanarini, "Counting-based search: Branching heuristics for constraint satisfaction problems," *J. Artif. Int. Res.*, vol. 43, no. 1, pp. 173–210, Jan. 2012.

P. Refalo, "Impact-based search strategies for constraint programming," in *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 557–571.

C. Schulte and P. J. Stuckey, "Dynamic analysis of bounds versus domain propagation," in *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December*

9-13 2008, Proceedings, ser. Lecture Notes in Computer Science, M. G. de la Banda and E. Pontelli, Eds., vol. 5366. Springer, 2008, pp. 332–346. [Online]. Available: https://doi.org/10.1007/978-3-540-89982-2_32

——, "When do bounds and domain propagation lead to the same search space?" in *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy.* ACM, 2001, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/773184.773197

R. Soto, B. Crawford, E. Monfroy, and V. Bustos, "Using autonomous search for generating good enumeration strategy blends in constraint programming," in *Computational Science and Its Applications - ICCSA 2012 - 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part III*, ser. Lecture Notes in Computer Science, B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds., vol. 7335. Springer, 2012, pp. 607–617. [Online]. Available: https://doi.org/10.1007/978-3-642-31137-6_46

K. Stergiou, "Heuristics for dynamically adapting propagation," in *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, ser. Frontiers in Artificial Intelligence and Applications, M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, Eds., vol. 178. IOS Press, 2008, pp. 485–489. [Online]. Available: https://doi.org/10.3233/978-1-58603-891-5-485

G. Tack, "Constraint propagation – models, techniques, implementation," Doctoral Dissertation, Saarland University, Jan. 2009.

H. Terashima-Marín, J. C. Ortiz-Bayliss, P. Ross, and M. Valenzuela-Rendón, "Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems," in *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, C. Ryan and M. Keijzer, Eds. ACM, 2008, pp. 571–578. [Online]. Available: http://doi.acm.org/10.1145/1389095.1389206

L. G. Valiant, "The complexity of computing the permanent," *Theoretical Computer Science*, vol. 8, no. 2, pp. 189 – 201, 1979.

W.-J. van Hoeve and I. Katriel, *Handbook of Constraint Programming.* Elsevier B. V., ch. 6, pp. 169–208.

T. Walsh, "019: Magic squares and sequences," March 2018. [Online]. Available: http://csplib.org/Problems/prob019/

R. J. Woodward, A. Schneider, B. Y. Choueiry, and C. Bessiere, "Adaptive parameterized consistency for non-binary csps by counting supports," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, ser. Lecture Notes in Computer Science, B. O'Sullivan, Ed., vol. 8656. Springer, 2014, pp. 755–764. [Online]. Available: https://doi.org/10.1007/978-3-319-10428-7_54

A. Zanarini, "Exploiting global constraints for search and propagation," Ph.D. dissertation, École Polytechnique de Montréal, 2010.

A. Zanarini and G. Pesant, "Solution Counting Algorithms for Constraint-Centered Search Heuristics," in *CP*, ser. Lecture Notes in Computer Science, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 743–757.

——, "More robust counting-based search heuristics with alldifferent constraints," in *CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, ser. Lecture Notes in Computer Science, A. Lodi, M. Milano, and P. Toth, Eds., vol. 6140. Springer, 2010, pp. 354–368.

ANNEX A NOTES ABOUT COUNTING ALGORITHMS IMPLEMENTATION

Distinct

In Gecode, there are three different propagators for the `distinct` constraint:

- `Gecode::Int::Distinct::Val`
- `Gecode::Int::Distinct::Bnd`
- `Gecode::Int::Distinct::Dom`

Each propagator corresponds to a different consistency level. All three propagators are defined in *gecode/int/distinct.hh*, with the implementations in the following files:

- *gecode/int/distinct/val.hpp*
- *gecode/int/distinct/bnd.hpp*
- *gecode/int/distinct/dom.hpp*

All three overloads of `solndistrib` and `domainsizesum` call the same two functions defined in *gecode/int/distinct/cbs.hpp*. When adding a new file in Gecode, it is important to modify *Makefile.in* accordingly, run *configure* and maybe *make depend*.

Regular

The regular constraint — instantiated by calling `extensional` in the user model — corresponds to the propagator `LayeredGraph` defined in *gecode/int/extensional.hh* and implemented in *gecode/int/extensional.hpp*. Unlike the previous propagators for the distinct constraint, the variables are located in the member variable `Layer* layers`, where `LayeredGraph ::Layer` is a subclass that encapsulate a variable.

Linear

In Gecode, there are multiple propagators for the `linear` constraint. We overloaded:

- `Gecode::Int::Linear::Eq`
- `Gecode::Int::Linear::Lq`

The definitions of those propagators are in *gecode/int/linear.hh*, with both implementations in *gecode/int/linear/int-nary.hpp*. Both overloads of `solndistrib` and `domainsizesum` call the same two functions defined in *gecode/int/linear/cbs.hpp*.

Generic Counting Algorithm Template for a New Constraint

First, we have to find the corresponding propagators for a constraint. It won't necessarily be a one-to-one correspondence; different propagators may be instantiated depending on the consistency level or the cardinality of the constraint.

The next step is to overload both methods introduced in Section 4.2.3 and 4.2.4. A generic template is given in Fig. A.1 and A.2.

```
/**
 * Let's suppose Prop contains its variables in:
 * - ViewArray<View> x;
 *
 * And let's suppose we already computed solution densities for our
 * hypothetical propagator "Prop" with a specialized algorithm
 * and that we can access them with dens[var_id][val].
 */
template<class View>
void Prop<View>::solndistrib(Space& home, Propagator::SendMarginal send)
{
    // For each variable in the propagator's view array
    for (int i = 0; i < x.size(); i++) {
        // If the variable is still not fixed
        if (viewArray[i].assigned()) continue;
        // For each value in the variable
        for (ViewValues<View> val(x[i]); val(); ++val) {
            send(id(),                          // Propagator ID
                 x[i].id(),                     // Variable ID
                 x[i].baseval(val.val()),       // Base value
                 dens[x[i].id()][val.val()] // Density
            );
        }
    }
}
```

Figure A.1 Generic template for a counting algorithm in `solndistrib`

```
/**
 * Let's suppose Prop contains its variables in:
 * - ViewArray<View> x;
 */
template<class View>
void cbssize(const ViewArray<View>& x, Propagator::InDecision in,
             unsigned int& size, unsigned int& size_b)
{
    size = 0;
    size_b = 0;
    // For each variable in the propagator's view array
    for (int i = 0; i < x.size(); i++) {
        // If the variable is not fixed
        if (!x[i].assigned()) {
            // We add the size of its domain to "size"
            size += x[i].size();
            // If it is included in the calling brancher, we
            // also add the size of its domain to "size_b"
            if (in(x[i].id())) size_b += x[i].size();
      }
    }
}
```

Figure A.2 Generic template for `domainsizesum`

ANNEX B FEATURES ANALYSIS FOR LOGISTIC REGRESSION

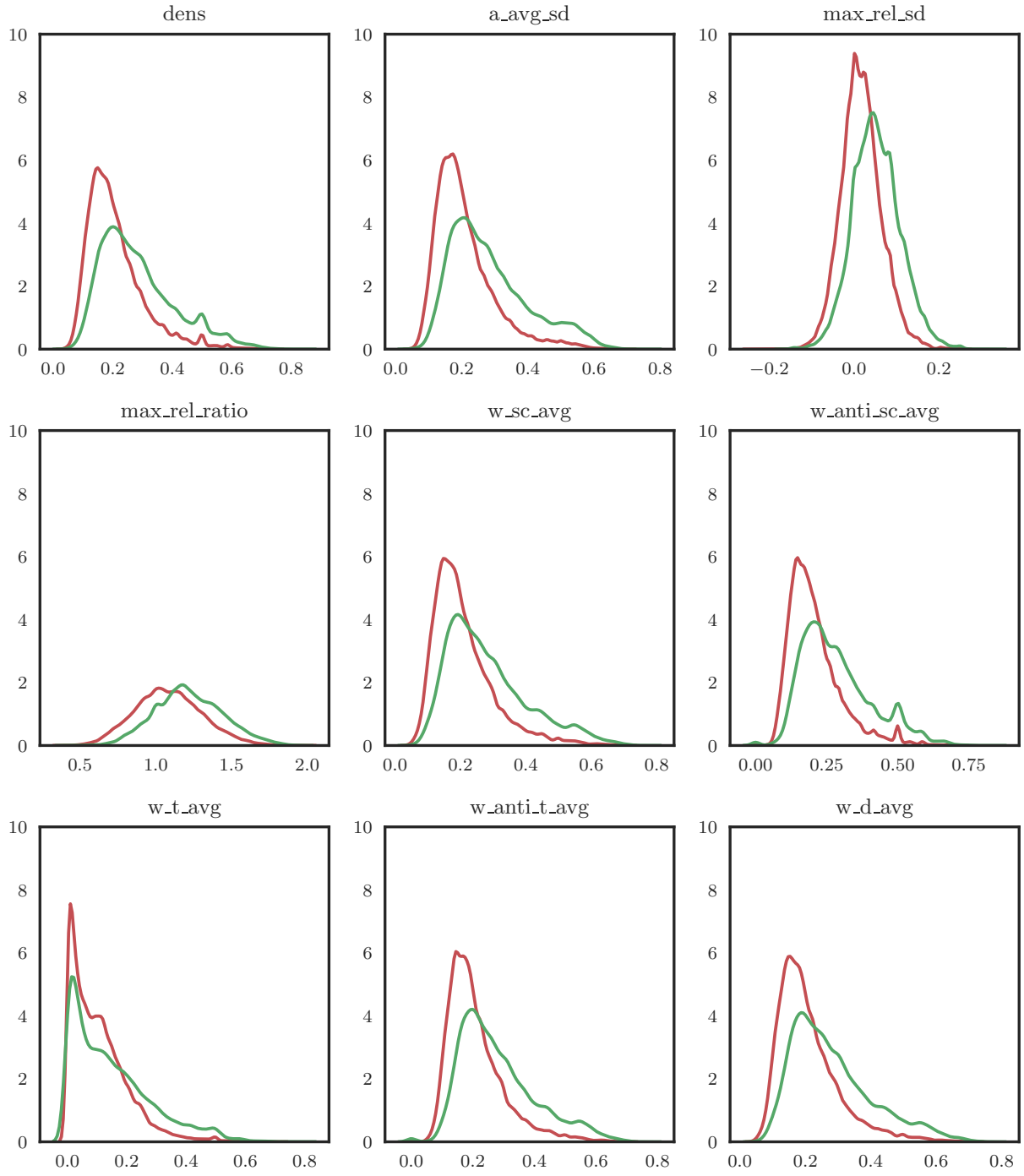


Figure B.1 Good and bad branching choices according to each branching heuristic