NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES
COMPUTER SCIENCE

MASTER THESIS

Extension and evaluation of the global cardinality constraints functionality of the Gecode open source toolkit

Ioannis A. Papatsoris

Supervisor: **Panagiotis Stamatopoulos,** Assistant Professor

ATHENS

AUGUST 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΛΗΡΟΦΟΡΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επέκταση και αξιολόγηση των καθολικών περιορισμών πληθικότητας στην πλατφόρμα ανοιχτού κώδικα Gecode

Ιωάννης Α. Παπατσώρης

Επιβλέπων: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΑΥΓΟΥΣΤΟΣ 2022

MASTER THESIS

Extension and evaluation of the global cardinality constraints functionality of the Gecode open source toolkit

Ioannis A. Papatsoris
R.N.: CS3180006

August 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επέκταση και αξιολόγηση των καθολικών περιορισμών πληθικότητας στην πλατφόρμα ανοιχτού κώδικα Gecode open source toolkit

Ιωάννης Α. Παπατσώρης
R.N.: CS3180006

August 2022

ABSTRACT

Constraint Programming is an Artificial Intelligence methodology that aims to solve real world problems in an efficient way. In this work, we extend the open source constraint solver Gecode by expanding its features concerning Global Constraints, specifically Global Cardinality Constraints. A Global Cardinality Constraint restricts the value occurrences among a collection of variables, to be between certain bounds. We develop the Global Cardinality Constraint With Costs, which is similar to the Global Cardinality Constraint and additionally associates a cost with each variable-value assignment, while further restricting the sum of the costs related to the assigned variable-value pairs to not exceed a given cost bound. Moreover, we add the Symmetric Global Cardinality Constraint, which is defined on Set variables and introduces additional restrictions on the cardinality of each set, aside from the value occurrences. We attempt to optimize their performance by experimenting with various different implementation choices, and finally we evaluate our constraints to discover under which conditions they are beneficial compared to decomposing them to multiple simpler ones.

SUBJECT AREA: Constraint Programming

KEYWORDS: constraint solver, open source, gecode, global constraint, global cardinality

ΠΕΡΙΛΗΨΗ

Ο Προγραμματισμός με Περιορισμούς είναι μια μεθοδολογία της Τεχνητής Νοημοσύνης που αποσκοπεί να επιλύσει πραγματικά προβλήματα με αποτελεσματικό τρόπο. Σε αυτή την διπλωματική εργασία, επεκτείνουμε τον επιλυτή προβλημάτων ικανοποίησης περιορισμών ανοιχτού κώδικα Gecode, συνεισφέροντας στις δυνατότητές του σχετικά με Καθολικούς Περιορισμούς, συγκεκριμένα περιορισμούς Global Cardinality. Ένας Global Cardinality περιορισμός περιορίζει τον αριθμό εμφάνισης τιμών μέσα σε μια συλλογή μεταβλητών, ώστε να βρίσκεται μεταξύ συγκεκριμένων ορίων. Αναπτύσσουμε τον περιορισμό Global Cardinality With Costs, ο οποίος είναι παρόμοιος του Global Cardinality και επιπλέον συσχετίζει ένα κόστος με κάθε ανάθεση τιμής σε μεταβλητή, ενώ ταυτόχρονα απαιτεί το άθροισμα των κοστών να μην ξεπερνάει ένα όριο. Στη συνέχεια προσθέτουμε τον περιορισμό Symmetric Global Cardinality, ο οποίος ορίζεται πάνω σε μεταβλητές που αφορούν σύνολα, δίνοντας επιπλέον περιορισμούς γύρω από τον πληθικό αριθμό του κάθε συνόλου, πέραν των περιορισμών που αφορούν τις τιμές. Ερευνούμε τη βελτιστοποίηση της επίδοσής τους, πειραματιζόμενοι με διάφορες εναλλακτικές επιλογές υλοποίησης, και τελικά τους συγκρίνουμε ώστε να ανακαλύψουμε κάτω από ποιές συνθήκες είναι ωφέλιμοι, σε σχέση με την αποσύνθεσή τους σε περισσότερους απλούστερους περιορισμούς.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Θεωρία Γράφων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: επιλυτής προβλημάτων ικανοποίησης περιορισμών, ανοιχτός κώδικας, gecode, καθολικός περιορισμός, global cardinality

ACKNOWLEDGMENTS

I would like to thank Professor Panagiotis Stamatopoulos for his sharing of knowledge through teaching along the years, and for his support and contribution to the realization of this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

At the early stages of this thesis, Gecode's features were compared with other constraint solvers to discover ideas and possible ways to extend its functionality and usability. After that it was decided that global cardinality constraints should be the point of focus, and literature research was conducted to conclude the specific constraints to implement. Gecode was studied through its documentation and by example, to get familiar with its internal environment and the methodology of programming custom constraints for it. While initially it was simple working versions of the constraints that were implemented, further on effort was focused into optimizing them by trying different implementation ideas, data structures and algorithms. Finally, experiments were conducted with computer generated and real world data for applications of the constraints, and they were compared to different configurations of their own and to using default constraints that already come with Gecode.

1. INTRODUCTION

In recent years, a lot of research interest has been focused on artificial intelligence. Constraint programming is an approach to represent and solve a particular problem, by modeling it with variables which can take values from a domain, and imposing constraints over them, in a way that mathematically expresses the nature of the problem. By assigning values to the variables while at the same time respecting the constraints, we can find a solution to it. To speed up the process, we can enhance our constraints to detect inconsistent values, and remove them from the domain before attempting to assign them. Many other strategies can be used with the goal to minimize execution time and solve problems faster, like deciding on which variables and values we should prioritize. There exist platforms called constraint solvers, that provide the necessary tools and environment to achieve this and make it accessible to programmers.

An important category of constraints is global constraints. Such constraints can involve a large non-fixed number of variables, and can be semantically represented by the conjunction of other simpler constraints. They are a key point of interest, because they can provide a more accurate view of the problem to the solver, which in turn can achieve deeper value propagation. Certain inconsistent values can be detected and pruned early from the solution, only if we treat the constraint as a global one, instead of breaking it down to multiple simpler ones.

A state-of-the-art constraint solver is Gecode [1], a powerful open source platform natively implemented in C++, that provides extensive features and customization to develop constraint satisfaction problems efficiently. In this work, we enhance the global constraints selection of the Gecode library, specifically focusing on global cardinality constraints. Particularly, we implement the Global Cardinality Constraint With Costs and the Symmetric Global Cardinality, both of which are absent from Gecode, as it includes only the regular Global Cardinality Constraint.

The Global Cardinality Constraint restricts the value occurrences assigned among a collection of variables, to be between a lower bound and an upper bound, potentially different for each value. The Global Cardinality Constraint With Costs further associates a cost with each variable-value assignment, and introduces the restriction that the sum of the costs related to a solution should not exceed an upper limit. The Symmetric Global Cardinality Constraint is similar to the original Global Cardinality but defined on set variables, which are variables whose domain is a set of sets of values. This version additionally limits the cardinality of the set of each variable, aside from the value occurrences. These constraints have already been proposed in the literature in [10], [2] and [18] respectively. The regular Global Cardinality Constraint can be viewed as a specialization of Global Cardinality Constraint With Costs in which all the costs are equal to 0, and as a specialization of Symmetric Global Cardinality where the cardinality bounds of each variable are equal to

1.

The publications around them are complete in terms of presenting and analyzing an efficient algorithm, but they do not provide guidance around implementation details. As we aim to not only realize them but also optimize their performance, we mainly experiment with the following ideas for the Global Cardinality Constraint With Costs:

- Improving the shortest paths search by keeping some costs of the residual graph negative and using Bellman-Ford's algorithm, instead of following the proposed reduced costs method that transforms them to positive in order to utilize Dijkstra's algorithm.
- Deleting edges from the value network as values get pruned with a backtrack-efficient structure that minimizes copying overhead during branching, which is suggested in [14], versus marking them as deleted but copying the whole graph each time.
- Not backtracking the flow versus backtracking it.
- A trivial data structure to hold the residual graph versus a more sophisticated one.
- A useful branching heuristic derived internally from the constraint, that prioritizes branching on values that are known to lead to a solution.

The value network is a graph that represents the constraint and consists of a node for each variable and value, and an edge from a value to a variable if the value belongs to the variable's domain, with some additional nodes and edges. The residual graph is derived from the value network and is used to gradually build a solution by assigning values to variables step by step, while it additionally expresses alternative decisions that can be made, or the ability to retract a choice. The term flow refers to a property of each edge which marks whether the respective variable and value associated with it are currently assigned to each other, and finding shortest paths along the residual graph is a way to choose the edges to provide flow with, so that the total cost will be minimized. These terms are explained in more detail in Chapter 2.2.2.

Additionally, we study how the constraint can be used in optimization problems, where the goal is not just to find any solutions lower than a given cost bound, but to minimize a cost function.

The Symmetric Global Cardinality Constraint algorithm shares a common core with the Global Cardinality Constraint With Costs, therefore we take the most efficient ideas that we described for the former and adapt them to the latter as well. Furthermore, we attempt an optimization mentioned in [14] that partitions edges of the graph to important and unimportant, and does not trigger the constraint on the removal of unimportant ones, as they would not cause any domain pruning.

The rest of the thesis is organized as follows:

1. In Chapter 2 we go through necessary knowledge around constraint programming,

global cardinality constraints, flow theory and constraint solvers, in order to build a foundation for the reader to be able to comprehend the rest of the chapters.

2. In Chapter 3 we focus on the Global Cardinality Constraint With Costs, and describe our implementation along with various alternative choices, a specialized branching heuristic, and we study its application as an optimization problem.
3. In Chapter 4 we discuss the Symmetric Global Cardinality Constraint, and mention our implementation and different ideas for it.
4. In Chapter 5 we conduct experiments on our constraints with real world and computer generated data, to discover which configuration of them performs best, and under which conditions they perform better than a decomposition of them.
5. In Chapter 6 we summarize our results and contribution.

2. BACKGROUND

2.1 Constraint Programming

Constraint programming is based on the idea that many interesting and difficult problems can be expressed declaratively in terms of variables and constraints. The variables range over a set of values and typically denote alternative decisions to be taken. The constraints are expressed as relations over subsets of variables and restrict feasible value combinations for the them. A solution is an assignment of variables with values which satisfies all constraints.

A key difference between the common primitives of imperative programming and constraint programming, is that in the latter we express the properties of the solutions that we are looking for, rather than specifying a sequence of algorithmic steps to execute. This means that certain problems may be intuitive to model as constraint satisfaction problems (CSPs), while others could be completely unsuitable for constraint programming. A classic example that can be well defined as a CSP is the N-Queens problem. In this problem, the goal is to place N queens on an $N \times N$ chessboard, such that no queen is attacking another. A possible way of modeling this problem as a CSP is the following:

1. Define N variables, representing a queen on each column.
2. Define the domain of each variable as $\{1, 2, ..., N\}$
3. Constrain $X_j \neq X_k$ for $j < k$
4. Constrain $|j - k| \neq |X_j - X_k|$ for $j < k$

We give an explanation for the steps above:

1. Since we have a variable for each column, it means that by default no queens will be attacking each other vertically.
2. The value of a variable X_j represents the row on which the queen for the column j will be placed.
3. Constraining all variables to have different values with each other ensures that no two queens will ever be placed on the same row.
4. Constraining the queens to not be the same number of columns apart as they are rows apart, ensures that no queen will ever attack another one diagonally.

Any particular problem can be represented in different ways, and the way we model it can make a tremendous difference in the size of the solution search space, yielding different performance results. For instance, a perhaps less optimized representation of N-Queens could be to have a boolean variable for all $N \times N$ positions on the chessboard, signifying whether we place a queen on them or not.

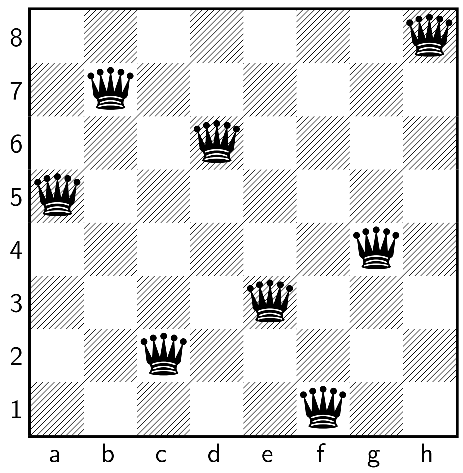

Figure 1: A solution to the 8-Queens problem

Constraint programming allows us to find feasible solutions for problems, solutions whose quality fits a certain criteria, or even "optimal" solutions. This could be a challenging task with regular programming techniques, as many of the problems are NP-Hard, meaning that we are not aware of a performant algorithm to solve them. Notable applications of constraint programming include crew scheduling, timetable creation, resource management, car sequencing, protein structure prediction and many others.

2.1.1 Notation

More formally, we can define the following notation involving around CSPs, which we will use throughout the thesis to describe algorithms and constraints. The following definitions are due to [2].

A finite constraint network $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, ..., x_n\}$, a set of current domains $\mathcal{D} = D(x_1), ..., D(x_n)$ where $D(x_i)$ is the finite set of possible values for variable x_i, and a set of **constraints** between variables. We introduce the particular notation $D_0 = \{D_0(x_1), ..., D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$. Indeed, we consider that any constraint network $\mathcal{N}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, ..., x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times ... \times D_0(x_{i_r})$ that specifies the allowed combinations of values for the variables $x_{i_1}, ..., x_{i_r}$ An element of $D_0(x_{i_1}) \times ... \times D_0(x_{i_r})$ is called a tuple on $X(C)$. $|X(C)|$ is the arity of C.

A value a for a variable x is often denoted by (x, a). **var**(C, i) represents the ith variable of $X(C)$, while **index**(C, z) is the position of variable x in $X(C)$. $\tau[k]$ denotes the kth value of the tuple τ. $D(X)$ denotes the union of domains of variables of X. $\#(a, \tau)$ is the number of occurrences of the value a in the tuple τ.

Let C be a constraint. A tuple τ on $X(C)$ is valid if $(x, a) \in \tau, a \in D(x)$. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is consistent with C iff

$x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $a = [\textbf{index}(C, x)]$. A constraint is **arc consistent** (or **domain consistent**) iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

A *bound support* on a constraint C is an assignment of all variables in the scope of C to values between their minimum and maximum values (called lower and upper bound respectively), such that C is satisfied. A variable-value $X_i = v$ is *bounds consistent* on C iff it belongs to a bound support of C. A constraint C is **bounds consistent** iff the lower and upper bounds of every variable in its scope are bounds consistent on C. A constraint is **range consistent** iff every value in the domain of every variable in the scope of C is bounds consistent on C.

2.1.2 Filtering

Not all combinations of variables and values necessarily form a solution. Constraints often attempt to remove inconsistent values, for which they can infer with certainty that they cannot participate in any solution. It is NP-Hard to decide whether a value is useful for the whole CSP at once, so normally this filtering is done locally in each constraint separately.

There is a distinction between *complete filtering*, which prunes all inconsistent values from variables involved in a constraint, and *partial filtering*, which removes only some of them. It is not always clear whether complete or partial filtering is preferred, because there is a trade-off between the effectiveness of the filtering (how many inconsistent values are removed) versus its efficiency (how long it takes to execute). The methods used to achieve this filtering depend on the nature of each constraint, and can range from trivial to understand inference, to complex dedicated algorithms.

Complete filtering is also known as *arc consistency* (or *domain consistency*). A variable is arc consistent with another one, if for each possible value assignment to it, there exists at least one admissible value for the other, according to the constraints that surround them. A CSP is arc consistent if every variable is arc consistent with each other. Partial filtering methods can include *bounds consistency* and *range consistency*, with the latter being stronger than bounds consistency, but still weaker than arc consistency.

Consider variables X and Y with domains $\{5, 6\}$ and $\{4, 7\}$ respectively, and the constraint $X < Y$. First and foremost, the constraint is consistent, because there is at least one possible assignment that forms a solution ($X = 5, Y = 7$). However, not all values individually are consistent. If we assign X with 5 or 6, there is at least one value in Y for which the constraint stands (the value 7). But if we assign Y with 4, there are no values in X for which $X < 4$, thus 4 is inconsistent and can be removed from the domain of Y. The value 7 for Y is also consistent. So after pruning 4 from Y, we have achieved arc consistency for this constraint.

Each time a variable is updated due to the filtering invoked by a constraint, all other relevant constraints to this variable are activated and checked again in order to verify consistency, and to potentially infer more filtering. This process is called *constraint propagation*, and is repeated until a fixed point is reached, meaning that no further reasoning can take place with the current state of the domains of the variables.

2.1.3 Search

Although constraint propagation can verify the consistency of constraints and keep the CSP up to date by removing inconsistent values, most of the time it is not enough to find solutions to non trivial problems. Thus we also need to assign values to variables in a systematic way. We call this procedure *search*. A naive way to search would be to enumarate all possible assignments for values to variables and then test against the constraints, to either declare solutions, or conclude that no more exist by exhausting the search space. Because such a method has exponential complexity in the best case, we combine it with constraint propagation, removing inconsistent values from the CSP every time a domain is altered. As a result, we can detect ”dead ends” early, before trying useless assignments.

We can think of search as a tree structure. It starts with a root node, and it branches from it using a branching strategy. An example of a simple one is to branch on a variable-value assignment, by creating two alternatives, one in which a specific variable is assigned to a specific value, and the alternative in which it is pruned from the domain of the variable. This procedure continues recursively, and on each step the consistency of the constraints involving the affected variables are verified, in addition to performing constraint propagation and potentially pruning values. If a failure is detected, the search backtracks and tries an alternative assignment, otherwise if all variables have been assigned, a solution is reported and then backtracking occurs to look for alternative solutions.

The choice on which variable to choose to branch on first and which values to prioritize for them can be impactful. One strategy is the *minimum remaining values* (MRV), according to which the variable with the least values is selected, thus the most restricted one. The intuition is that since this variable is the most likely to cause a failure, and that we would have to assign it eventually, it’s better to do it sooner than later and prevent pointless assignments to other variables before it. If we are looking for just one solution and not all of them, a value selection strategy that can be beneficial is *least constraining values* (LCV), which tries to avoid failure by assigning values that allow for maximal flexibility for the remaining variables, aiming for a branch that is most likely to succeed.

Since every problem has unique characteristics, it is necessary to experiment with different branching strategies to discover the most performant ones. According to the nature of the problem, quite often it is possible and advised to use a dedicated branching strategy specifically adapted to the problem itself, to maximize performance.

2.1.4 Global Constraints

In the ealier days of constraint programming, research was initially focused in designing efficient filtering algorithms for fundamenantal constraints that are common and can be adopted by any problem, like simple value relations ($=, \neq, \leq, \geq, ...$). As time went by, it became more and more evident that for more complex problems, it can be inconvenient to express complicated constraints just by combining the basic ones. Aside from ease of design, it was also observed that domain filtering was limited to the local scope of these simple constraints, and was unable to examine the whole picture of the problem. Thus attention started to focus on a new category of constraints, named *global constraints*.

Global constraints encapsulate the a set of other simpler constraints. We can see an example of the driving motivation behind them in the constraint ALLDIFF, which requires all variables to take a different value from each other. This constraint is equivalent to pair-wise inequality constraints ($\neq$) between each variable. Consider variables x_1, x_2, x_3 with domains $D(x_1) = \{a, b\}, D(x_2) = \{a, b\}, D(x_3) = \{a, b, c\}$. Without the global constraint, we would achieve the desired effect by constraining $x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3$. Enforcing arc consistency here would not prune any value, however the arc consistency of the global constraint ALLDIFF would remove values a and b from x_3.

We present a summary of categories and examples of global constraints, drawing information from the survey in [3].

We can distinguish global constraints in the following categories:

- **Classical Constraints.** Common constraints like ALLDIFF, GCC, REGULAR, SEQUENCE, PATH...
- **Weighted Constraints.** Constraints that are associated with some form of cost or weight. A lot of NP-Hard problems are on this category, like COSTGCC, KNAPSACK, BINPACKING...
- **Soft Constraints [4].** They are relaxed versions of classical or weighted constraints. They often come with an additional cost variable measuring the distance to the full satisfaction.
- **Constraints on Meta-Variables [5].** These are constraints defined on set and graph variables, instead of classical ones.
- **Open Constraints [6].** In this category, the exact variables involved in the constraint are not known. Instead, we only know variables which could *potentially* be involved.

For the first two categories, we list the following subcategories of global constraints, along with some examples.

- **Counting Constraints.** ALLDIFF, PERMUTATION, global cardinality (GCC), global cardinality with costs (COST-GCC), cardinality matrix constraints (CARD-MATRIX)

- **Balancing Constraints.** BALANCE, DEVIATION, SPREAD
- **Combination based Constraints.** MAX-SAT, OR, AND
- **Sequencing Constraints.** AMONG, SEQUENCE, generalized sequence GEN-SEQUENCE, global sequencing constraints (GSC).
- **Distane Constraints.** INTER-DISTANCE, SUM_INEQ
- **Geometric Constraints.** DIFF-N
- **Summation based Constraints.** SUBSET-SUM, KNAPSACK
- **Packing Constraints.** SYM-ALLDIFF, STRETCH, K-DIFF, number of distinct values (NVALUE), BIN-PACKING
- **Graph based Constraints.** CYCLE, PATH, TREE, weighted spanning tree (WST)
- **Order based Constraints.** LEXICO$\leq$, SORT

2.2 Global Cardinality

The global cardinality constraint (GCC) belongs in the *counting* category of global constraints, and it constrains the number of times every value can be taken by a set of variables. Specifically, it restricts the frequency of each value to be within a certain range, which can differ for each one. The ALLDIFF constraint is a specialization of GCC in which the ranges are $[0, 1]$, thus every value can be used at most once.

For a formal definition, we have the following:

A **global cardinality constraint** is a constraint C in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i and

$$T(C) = \Big\{\tau \text{ such that } \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\Big\}$$

It is denoted by gcc(X, l, u).

The GCC constraint appears in many scheduling and rostering problems. We mention some real world application examples of it.

- **Sports scheduling:** The problem is introduced in [7]. It involves scheduling games between n teams over $n-1$ weeks, with each week being divided into $n/2$ periods. The following constraints must be met:

  1. Each team must play against every other team.
  2. A team must play exactly once a week.
  3. A team must play at most twice in the same period, across the season.

  The third constraint is achieved with GCC.

- **Car sequencing:** The car sequencing problem [8] is to sequence cars on a conveyor through a factory. There are a number of optional parts that may be fitted to the cars, and each optional part has a corresponding machine which fits the part. For an option i, the machine cannot accept more than p_i cars in every q_i . Therefore, in every contiguous subsequence of length q_i there must be no more than p_i cars requiring the option. There are a number of different types of car, where each type has a set of options that it requires, and a fixed number of each type is required in the sequence. Multiple GCCs are used to enforce these restrictions.

- **Equidistant frequency permutation arrays (EFPAs):** The EFPA problem [9] is to find a set (often of maximal size) of code words, such that any pair of codewords are Hamming distance d apart. Each code word is made up of symbols from the alphabet $\{1, ..., q\}$, with each symbol occurring a fixed number λ of times per code word. A fourth parameter v is the number of code words in the set. Typically v would be maximized. Multiple GCCs are utilized to enforce λ occurrences of each symbol.

In GCC, the lower and upper bounds are fixed values. It is worth mentioning that we can define a generalized version of GCC called the extended global cardinality constraint (EGCC) in which the cardinality values are not constant, but they appear as variables, being able to take different values. An example problem of EGCC is the Magic Sequence problem, where the goal is to find a sequence of given length n such that element i in the sequence is the number of occurrences of i in the sequence.

2.2.1 Related Work

Since GCC is a fundamenantal constraint which appears in many different real world applications, naturally several variations of it have been proposed and studied. We start off by mentioning the state-of-the-art research in the literature for the classic GCC, and after we list some adaptations.

- **GCC Arc Consistency by Régin:** In [10], an algorithm based on flow theory is introduced, running in $O(|X(C)|^2|D(X)|)$ and achieving arc consistency in $O(\delta + |X(C)| + |D(C)|)$, where δ is the number of arcs in the value network of the constraint (explained more in publication). To check the consistency of the constraint, it uses Ford-Fulkerson's algorithm [11] to compute a flow which corresponds to an assignment to the target variables, satisfying the lower and upper bounds for each value. Afterwards, Tarjan's algorithm [12] is used for the search of strongly connected components, to compute the set of edges that cannot belong to any maximum flow. These edges correspond to the domain values to be pruned. Ford-Fulkerson's algorithm time complexity dominates the algorithm.

- **GCC Arc Consistency by Quimper:** In [13], an algorithm based on graph matching is presented with complexity $O(|X(C)|^{1.5}|D(X)|)$, improving upon Régin's one. In addition, a cardinality variable pruning algorithm is shown for the case of EGCC, which runs in $O(|X(C)|^2D(X) + |X(C)|^{2.66})$

- **Survey on GCC and EGCC arc consistency algorithms:** In [14], techniques and implementation optimizations are gathered from the above publications and other literature, along with some newly proposed ones, and they are combined and benchmarked together. Régin's flow based algorithm is upgraded to make use of such optimizations, achieving 4 times faster performance compared to an unoptimized version. It is observed that even though Quimper's algorithm has better complexity, in practice Régin's algorithm runs faster thanks to its simplicity.
- **GCC Bounds/Range Consistency:** In [15], a bounds consistency algorithm is described exploiting bipartite graph convexity and running in $O(|X(C)| + |D(C)| + t)$, where t is the time required to sort the assignment variables by range endpoints. It is special in the fact that it prunes the cardinality variables in addition to the assignment ones, making in compatible for EGCC. An improved algorithm in [16] which is specific to the simple GCC without cardinality variables, achieves bounds consistency in $O(|X(C)| + t)$, by identifying Hall intervals. This algorithm is later used in [17] to achieve range consistency in $O(X(C) + t + N)$, where N is the number of values with a null lower bound.

We now mention some global cardinality variations proposed in the literature. We can define COST-GCC, which is the same as the original global cardinality constraint, but with the addition that a cost is associated with every variable-value pair assignment. The added restriction is that the sum of the costs of the variable-value pairs which satisfy the cardinality constraints, should be lower or equal to a given bound. This constraint is proposed in [2] and solved using flow theory, and is discussed further in Chapter 3.

In [18], the *symmetric global cardinality* constraint is proposed (SYM-GCC), which is like the GCC but is defined on set variables. Thus, a set variable can take none, one, or multiple values from each domain. As an additional restriction to the value cardinalities, for each variable we restrict its set cardinality to be between a lower and upper bound, potentially different for each variable. This constraint is approached using flow theory as well. We look into it more in Chapter 4.

In [19], we can find *symmetric global cardinality with costs* SYM-COST-GCC, which is the weighted version of SYM-GCC, associating costs with each variable-value assignment and restricting the sum of the costs that appear in a solution to an upper bound, similarly to COST-GCC. Once more, flow theory is used to solve it.

In [20], an alternative method to flow theory utilizing graph matching is discussed, and different solutions are given to some of the above constraints, like COST-GCC, SYM-GCC and SYM-COST-GCC. Other global cardinality family constraints can also be found, like *symmetric alldifferent*, *alldifferent with costs*, *symmetric alldifferent with costs*.

In [6], the concept of *open constraints* is introduced, which refers to constraints that are not a priori defined on specific sets of variables, but their variables may be discovered during the solution process. This problem can arise often in scheduling applications and other distributed settings. The article deals specifically with open global cardinality constraints, and the conjunctions of them (in case they are defined on disjoint sets of variables), and provides a set-domain consistency algorithm, based on flow theory, along with a weaker propagation algorithm for the case when they not disjoint.

In [21], a constraint called the *ordered distribute* constraint is described, which restricts the number of times a value v or any value greater than v is taken by the variables. It is an extension of the global cardinality constraint, taking into account also the values greater than v. This constraint can be useful in solving assignment problems, where teams needs to be balanced in respect to hierarchical skills of the members, or in over-constrained problems, in which costs represent degrees of violation of constraints. A linear algorithm that achieves arc consistency is proposed.

In [22], the SAME constraint is extended with GCC-like restrictions. The SAME constraint is defined among two sets of variables, and it enforces that the multiset of the values assigned in the first, is equal to the multiset of the values assigned in the second. This new constraint adds lower and upper bound restrictions on the occurrences of the values, on top of the SAME constraint. It can be used to model certain scheduling problems. A flow based approach is presented for arc consistency, along with a faster bounds consistency algorithm for a restricted case of it.

2.2.2 Flow Theory

Since flow theory is fundamental for solving GCC constraints and has been used extensively in literature, we present some basic concepts of it, which we will refer to in the later sections of this thesis as well. The following are taken from [2], which in turn are based on [23, 24, 25, 26].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **vertex set** X and an **arc set** U, where every arc (u, v) is an ordered pair of distinct vertices. We will denote by $X(G)$ the vertex set of G and by $U(G)$ the arc set of G. The **cost** of an arc is a value associated with the arc.

A **path** from node v_1 to node v_k in G is a list of nodes $[v_1, ..., v_k]$ such that (v_i, v_{i+1}) is an arc for $i \in [1...k-1]$. The path **contains** node v_i for $i \in [1...k]$ and arc (v_i, v_{i+1}) for $i \in [1...k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $v_1 = v_k$. The **length** of a path p, denoted by *length(p)*, is the sum of the costs of the arcs contained in p. A **shortest path** from a node s to a node t is a path from s to t whose length is minimum. A cycle of negative length is called a **negative cycle**. Let s and t be nodes, there is a shortest path from s to t if and only if there exists a path from s to t and no path from s to t contains a negative cycle. If there is a shortest path from s to t, there is one that is simple.

The complexity of the search for shortest paths from a node to every node in a graph with m arcs and n nodes depends on the maximal cost γ and on the sign of the costs. Therefore, we will denoted this complexity by $S(m, n, \gamma)$ if all the costs are nonnegative; and $S_{neg}(m, n, \gamma)$ otherwise.

Let G be a graph for which each arc (i, j) is associated with three integers l_{ij}, u_{ij}, and c_{ij}, respectively called the **lower bound capacity**, the **upper bound capacity** and the **cost** of the arc.

A **flow** in G is a function f satisfying the following two conditions:

- For any arc (i, j), f_{ij} represents the amount of some commodity that can “flow” through the arc. Such a flow is permitted only in the indicated direction of the arc, i.e., from i to j. For convenience, we assume $f_{ij} = 0$ if $(i, j) \notin U(G)$.
- A **conservation law** is observed at each node: $\forall j \in X(G) : \sum_i f_{ij} = \sum_k f_{jk}$. The **cost** of a flow f is $cost(f) = \sum_{(i,j)\in U(G)} f_{ij} c_{ij}$.

We will consider three problems of flow theory:

- **the feasible flow problem**: Does there exist a flow in G that satisfies the **capacity constraint**? That is find f such that $\forall(i, j) \in U(G) l_{ij} \leq f_{ij} \leq u_{ij}$.
- **the problem of the maximum flow for an arc** (i, j): Find a feasible flow in G for which the value of f_{ij} is maximum.
- **the minimum cost flow problem**: If there exists a feasible flow, find a feasible flow f such that $cost(f)$ is minimum.

Without loss of generality, we will consider that:

- if (i, j) is an arc of G then (j, i) is not an arc of G.
- all boundaries of capacities are nonnegative integers.

Consider, for instance, that all the lower bounds are equal to zero and suppose that we want to increase the flow value for an arc (i, j). In this case, the flow of zero on all arcs, called the **zero flow**, is a feasible flow. Let P be a path from j to i different from (j, i), and $val = min(\{u_{ij}\} \cup \{u_{pq} s.t. (p, q) \in P\})$. Then we can define the function f on the arcs of G such that $f_{pq} = val$ if P contains (p, q) or $(p, q) = (i, j)$, and $f_{pq} = 0$ otherwise. This function is a flow in G. (The conservation law is obviously satisfied because (i, j) and P form a cycle.) We have $f_{ij} > 0$; hence it is easy to improve the flow of an arc when all the lower bounds are zero and when we start from the zero flow. It is, indeed, sufficient to find a path satisfying the capacity constraint. The main idea of the basic algorithms of flow theory, is to proceed by successive modifications of flows, that are computed in a graph in which all the lower bounds are zero and the current flow is the zero flow. This particular graph can be obtained from any flow and is called the residual graph:

The **residual graph** for a given flow f , denoted by $R(f)$, is the digraph with the same node set as in G. The arc set of $R(f)$ is defined as follows: $\forall(i, j) \in U(G)$:

- $f_{ij} < u_{ij} \Leftrightarrow (i, j) \in U(R(f))$ and has cost $rc_{ij} = c_{ij}$ and upper bound capacity $r_{ij} = u_{ij} - f_{ij}$.
- $f_{ij} > l_{ij} \Leftrightarrow (j, i) \in U(R(f)$ and has cost $rc_{ji} = -c_{ij}$ and upper bound capacity $r_{ji} = f_{ij} - l_{ij}$.

All the lower bound capacities are equal to 0. Instead of working with the original graph G, we can work with the residual graph $R(f^0)$ for some f^0. From f' a flow in $R(f^0)$, we can obtain f another flow in G defined by $\forall(i, j) \in U(G) : f_{ij} = f^0_{ij} + f'_{ij} - f'_{ji}$. And from a path in $R(f^0)$ we can define a flow f' in $R(f^0)$ and so a flow in G:

We will say that f is obtained from f^0 by sending k units of flow along a path P from j to i if:

- P is a path in $R(f^0) - \{(j, i)\}$
- $k = min(\{r_{ij}\} \cup \{r_{uv} s.t. (u, v) \in P\})$
- f corresponds in $R(f^0)$ to the flow f' defined by:
  - $f'_{pq} = k$ for each arc $(p, q) \in P \cup \{(i, j)\}$
  - $f'_{pq} = 0$ for all other arcs.

Let f^0 be any feasible flow in G, and (i, j) be an arc of G.

- There is a feasible flow f in G with $f_{ij} > {f^0}_{ij}$ if and only if there exists a path from j to i in $R(f^0) - \{(j, i)\}$.
- There is a feasible flow f in G with $f_{ij} < {f^0}_{ij}$ if and only if there exists a path from i to j in $R(f^0) - \{(i, j)\}$.

Maximum Flow Algorithm: With the above, we can construct a maximum flow in an arc (i, j) by iterative improvement, due to Ford and Fulkerson [11]: Begin with any feasible flow f^0 and look for a path from j to i in $R(f^0) - \{(j, i)\}$. If there is none, f^0 is maximum. If, on the other hand, we find such a path P, then define f^1 obtained from f^0 by sending flow along P. Now look for a path from j to i in $R(f^1) - \{(j, i)\}$ and repeat this process. When there is no such path for f^k, then f^k is a maximum flow. A path can be found in $O(m)$, so a maximum flow of value v in an arc (i, j) can be found from a feasible flow in $O(mv)$.

Feasible Flow Algorithm: For establishing a feasible flow, follow this method which repeatedly searches for maximum flows in some arcs:
Start with the zero flow f^0. This flow satisfies the upper bounds. Set $f = f^0$, and apply the following process while the flow is not feasible:

1. pick an arc (i, j) such that f_{ij} violates the lower bound capacity in G (i.e. $f_{ij} < l_{ij}$).
2. Find P a path from j to i in $R(f) - \{(j, i)\}$.
3. Obtain f' from f by sending flow along P; set $f = f'$ and goto 1)

If, at some point, there is no path for the current flow, then a feasible flow does not exist.

Otherwise, the obtained flow is feasible.

Minimum Cost Flow Problem: The search for a feasible flow with a minimum cost implies only few modifications in the previous algorithm to ensure that the cost of the feasible flow will be minimum. In fact, only one aspect of the method is modified; the flow will be obtained by sending flow along special paths: the shortest ones. That is, the shortest paths are computed in the residual network by using the residual cost as cost. This algorithm is called the **successive shortest paths** algorithm.

Incrementality: Suppose f^0 is a minimum cost flow in G^0, and G is the same graph as G^0 except that some capacity boundaries have been tightened (i.e. some lowers bounds have been increased and some upper bounds have been decreased). f^0 is not necessarily feasible in G. We can obtain a feasible flow in G which is also a minimum cost flow or prove there is none by applying the following algorithm:
Start with $f = f^0$ and apply the following process while f is infeasible in G: Pick an arc (i, j) such that f_{ij} violates a bound capacity in G. If $f_{ij} < l_{ij}$, then find P a shortest path from j to i in $R(f) - \{(j, i\})$. If $f_{ij} > u_{ij}$, then find P a shortest path from i to j in $R(f) - \{(i, j\})$. Obtain f' from f by sending flow along P; set $f = f'$. If, at some point, there is no path for the current flow, then a feasible flow does not exist. Otherwise, the obtained flow is minimum cost flow.

2.3 Constraint Solvers

Constraint solvers are platforms which provide the necessary environment to successfully build, benchmark and study CSPs. They are typically packaged as a library to a specific programming language, and they consist of a toolbox that allows users to define CSPs declaratively, combining already built in constraints to form more complex ones. Search engines are implemented within a constraint solver, and the programmer can specify which configuration they would like to use for their application, through controllable parameters. They can offer a plethora of different branching and search strategies, which in turn allows for extensive experimental evaluation of a particular program and benchmarking. Several other features are common as well, like debugging conveniences, programming new constraints and customizations to suit an application's specific needs.

Constraint solvers originated as an extension of *Logic Programming*, creating the field of *Constraint Logic Programming* (CLP). Taking advantage of the declarative nature of logic programming, modelling problems with constraints has been intuitive within languages like Prolog, combined with constraint programming libraries like ECLiPSe [27]. However, logic programming itself did not apply to a wide audience. It's an unconventional paradigm, demanding a different way to think and approach problems compared to procedural programming, which can be challenging for beginners to understand and ease into. Thus later on effort was put into designing constraint solvers for different languages, so that constraint programming could become more accessible to a broader audience, and find itself in more industrial applications and promote research. On this thesis we will focus on the constraint solver Gecode, as it is the one our work is based on.

2.3.1 Gecode

Gecode is a state-of-the-art free open source environment for developing constraint-based systems and applications, implemented in C++. It offers a wide range of features and customization, allowing the programmer to extend almost every part of it. Its core is optimized for performance with respect to runtime, memory usage and scalability, in addition to exploiting multiple cores to allow for parallel search, and it has won multiple benchmarking awards. Finally, it has a sizeable and loyal user community that contributes to it frequently, an online group to ask and answer inqueries, and rich and extensive documentation to understand every part of it.

We take a closer look at some of the features of Gecode. For a complete and detailed view, please consult its official documentation.

In Gecode, we can create models using variables of types Integer, Boolean, Float, and Set. It is also possible to program new variable types, at the same efficiency level as the built-in ones. For each variable type, there exists a package of constraints that we can use to restrict their domain and design our model. Certain constraints can even allow the combination of different variable types (for example, Integer and Boolean).

For each constraint, we can define the propagation level that we desire, based on what are available. That is, we can use value propagation, bound consistency, or domain consistency. Not all constraints offer all the propagation levels, so we consult the relevant documentation for that. Furthermore, we can program our own constraints. The Gecode documentation includes a chapter with guidelines about how to build a constraint the correct way, and optimize it accordingly to avoid pointless propagator execution.

We can choose the desired way to branch on variables and values, from a plethora of predefined ones, but we can also create our own. This can be done either by providing a function to branch on variables and a function to choose the value to branch on, or by programming a completely new brancher from scratch, allowing for deeper control to achieve more sophisticated and targeted behavior. Gecode is packed by default with all the common variable-value branching strategies, in addition to including more advanced ones, like the following:

- **Accumulated Failure Count (AFC):** The AFC of a variable (also known as weighted degree) is defined as the sum of the AFCs of all propagators depending on the variable plus its degree (to give a good initial value if the AFCs of all propagators are still zero). The AFC of a propagator counts how often the propagator has failed during search.

- **Action:** The action of a variable captures how often its domain has been reduced during constraint propagation.
- **Conflict-History Based Branching (CHB):** The CHB of a variable combines how often its domain has been reduced during constraint propagation with how recently the variable has been reduced during failure.

In case of a tie during the selection, Gecode allows the programmer not only to specify tie breaking strategies, but to also manually precise what is considered as a tie.

In some problems, there exist many solutions who are essentially the same because they are symmetric. Gecode supports *Lightweight Dynamic Symmetry Breaking* (LDSB [**?**]), that is, given a specification of the symmetries, it can avoid visiting symmetric states during the search, which can result in dramatically smaller search trees and greatly improved runtime.

A requirement for solution search is that it can return to previous states, because an alternative suggested by a branching may not lead to a solution, or even if a solution has been found more solutions might be requested. It is vital to have a system in place that is optimized and efficient at its core, to be able to traverse through the solution space effectively. Gecode employs a technique called *hybrid recomputation*, along with an optimization named *adaptive recomputation*, to not slow down in situations where we are stuck in a failed subtree because of an incorrect choice higher up in the search tree.

Gecode supports by default 3 search engines: *depth-first left-most (DFS)*, *limited discrepancy* (LDS), *branch-and-bound* (BAB). DFS and BAB support parallel execution, to explore different parts of the search tree simultaneously. Every search engine can be configured by an extensive selection of parameters, and fully custom search engines can be programmed from scratch as well.

The *Graphical Interactive Search Tool* (Gist), provides user-controlled search, search tree visualization, and inspection of arbitrary nodes in the search tree. Gist can be helpful when experimenting with different branching strategies, with different models for the same prob lem, or with propagation strength (for instance bounds versus domain propagation). It gives direct feedback on how the search tree looks like, if the branching heuristic works, or where propagation is weaker than expected.

More and more possibilities exist, such as *restart-based search*, *portfolio search*, *no-goods*, *tracing*, *CPProfiler support*. Further information can be found on the Gecode documentation itself.

3. GLOBAL CARDINALITY WITH COSTS

The global cardinality constraint with costs (costgcc) restricts the minimum and maximum number of occurrences of each value just like the original global cardinality, with the addition of a cost associated with each variable-value assignment, and the constraint that the sum of the assigned costs should be less or equal to a given cost upper bound.

Costgcc can arise in scheduling applications. Consider an example derived from a real problem given in [28]. We need to schedule managers for a directory-assistance center, with 5 activities and 7 people over 7 days. Let's study only one particular day: a person has to perform an activity, and there can be a minimum and maximum number of times that this activity can be performed in general. Each person might have the technical skills to perform a different set of activities. This constraint can be expressed with a regular gcc. Now, if we were to include a preference value for each person-activity pair, and say that we would like the total preference value of everybody to be less than an upper bound in order to improve worker satisfaction, we can use costgcc. Instead of preference, the costs could also signify how unsuitable each person is for a particular activity. People with low cost are more suitable than those with high cost, and so we would like to bound the total value of unsuitability.

A more complex industrial problem where costgcc could be used is Continuous Casting Steel Production with Electricity Bill Minimization, as described in [29].

More formally, we define a **cost function on a variable set** X as a function which associates with each value $(x, a), x \in X$ and $a \in D(x)$ an integer denoted by $cost(x, a)$. A **global cardinality constraint with costs** is a constraint C associated with $cost$ a cost function on $X(C)$, an integer H and in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i

$$T(C) = \Big\{\tau \text{ such that } \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\} \text{ and } \sum_{i=1}^{|X(C)|} cost(var(C, i), \tau[i]) \leq H\Big\}$$

The code developed for this constraint, along with a usage example for the experiments conducted on Chapter 5.1.1 can be found in the Github repository in [30] under the branch name *costgcc*, in the subdirectory *gecode/int/cost-gcc*.

3.1 Modeling With Gecode

Costgcc can be decomposed into the conjunction of a gcc and a sum constraint. Gecode offers the original gcc through the constraint `count`. Consider $X = \{X_1, ..., X_n\}$ to be the set of variables involved in costgcc, $D = \{D_1, ..., D_m\}$ to be the set of different possible values, and C to be an integer array of n rows and m columns, with each element C_{ij} holding a cost value associated with assigning variable X_i with value D_j. We can model the sum part in the following way:

1. Define B as a boolean variable array of n rows and m columns.
2. Constrain each element B_{ij} of B to be true iff $X_i = D_j$, false otherwise.
3. Use `linear` constraint to restrict that the sum of each element of C multiplied by the respective element of B, will be less or equal to the given cost upper bound.

This decomposition can miss pruning some inconsistent values, as it cannot see the global picture involving the sum, and Gecode does not include costgcc, thus the motivation of this thesis for choosing to implement it.

3.2 Constraint Usage

The costgcc we have implemented is

```
countCosts(home, X, D, L, U, C, H, B, P)
```

as follows:

- **home**: the current Gecode home space. Mandatory argument for all Gecode constraints.
- **X**: array of the variables to constrain (type `IntVarArgs`).
- **D**: array of all the different possible domain values (type `IntArgs`).
- **L**: array containing the lower bound for each value in D (type `IntArgs`).
- **U**: array containing the upper bound for each value in D (type `IntArgs`).
- **C**: array containing the cost associated with each variable-value assignment (type `IntArgs`). Element $C[i \times |D| + j]$ corresponds to the cost of assigning variable $X[i]$ with value $D[j]$.
- **H**: cost upper bound. Is of type `IntVar` and its maximum value is regarded as the cost upper bound.
- **B**: Controls whether to use the custom branching technique described in 3.5 or not. Is of type `BestBranch *`, and in case we don't want to use it, it should be `NULL`. Optional argument, default value is `NULL`.
- **P**: optional argument. Specifies propagation level, accepts `IPL_DOM` for arc consistency or `IPL_VAL` for just checking if the constraint holds, but without pruning any values. Optional argument, default value is `IPL_DOM`.

The cost upper bound is given in the form of a Gecode variable, to be able to tighten it in case we use a Branch and Bound method to solve an optimization problem. There is no inference on the cost variable, each time only its maximum value is taken as the cost sum bound.

Using the argument B for the custom branching described in Chapter 3.5 is **highly recommended**, as we will show in Chapter 5 that it dramatically improves performance.

`BestBranch` is a class that implements a Local Object Handle. This is Gecode's way to share data structures between propagators and branchers. This class serves as the link between them, as it is written by the propagator and read by the brancher, to make a smart branching decision. More information can be found on the *Managing Memory* section of the *Programming Propagators* chapter in the Gecode documentation.

In addition to the `BestBranch` class, we also need to implement a custom brancher, which will use this information and branch accordingly. For example code of this class and the custom brancher, see Figures 12 and 13 respectively in the Annex.

An exception is thrown if one of the following conditions involved around the arguments is not met:

- X should not contain duplicate variables and should be of at least size 1.
- D should not contain duplicates.
- D should include all the values from the domains of the variables in X.
- L and U arrays should be the same size as D.
- Bounds in L and U should be non-negative.
- Each lower bound $L[i]$ should be smaller or equal to the respective upper bound $U[i]$.
- C should be of size $|X| \times |D|$.

There is no restriction on the sign of the costs and the cost upper bound. Cost values $C[ij]$ for which the value $D[j]$ does not belong to the domain of variable $X[i]$ are ignored.

3.3 Algorithm

The base algorithm that we implement proposed by Régin [2] is based on network flows. We first establish a foundation by studying the simple gcc without costs.

3.3.1 Global Cardinality Constraint

Given $C = gcc(X, l, u)$ be a gcc; we define the **value network** of C to be the directed graph $N(C)$ with a lower bound and upper bound capacity on each edge, as follows:

- for each variable v and value u that belongs to its domain, add an edge (u, v) with $l_{uv} = 0$ and $u_{uv} = 1$.
- add a node s and an edge from s to each value. For such an edge $(s, a_i) : l_{sa_i} = l_i$, $u_{sa_i} = u_i$.
- add a node t and an edge from each variable to t. For such an edge $(x, t) : l_{xt} = 1$,

$u_{xt} = 1$.

- add an edge (t, s) with $l_{ts} = u_{ts} = |X(C)|$.

To prove feasibility for gcc, we need to find a feasible flow in $N(C)$. A feasible flow corresponds to a legal solution which consists of the variable-value assignments that map to the edges (u, v) that end up having flow through them. The intuition behind this connection between a feasible flow in the value network and a legal solution to the constraint is the following:

- All variables will be instantiated to exactly one value, to satisfy the lower and upper bounds of 1 of the (v, t) edges.
- A value may or may not be assigned to a variable of its domain, thus the lower bound of 0 and upper bound of 1 for (u, v).
- The amount of flow through the (s, u) edges signifies the number of occurrences of said value u in the solution, and thus it will respect the corresponding lower and upper bounds specified by gcc for this value.

Let's take a look at an example. Consider the following variables with their respective domain values:

$$\begin{aligned} peter &= \{M, D\} \\ paul &= \{M, D\} \\ mary &= \{M, D\} \\ john &= \{M, D\} \\ bob &= \{N\} \\ mike &= \{B\} \\ julia &= \{B, O\} \end{aligned}$$

and a GCC that restricts the occurences of each value as follows:

$$\begin{aligned} l_M &= 1 \; u_M = 2 \\ l_D &= 1 \; u_D = 2 \\ l_N &= 1 \; u_N = 1 \\ l_B &= 0 \; u_B = 2 \\ l_O &= 0 \; u_O = 2 \end{aligned}$$

In Figure 2 we see a variable-value network for this particular instance, taken from [2], and with a solution already formed on it. Edge $M \to peter$ has flow because variable $peter$ is assigned to value M. According to the flow conservation law, the incoming flow amount of a node should be equal to its outgoing amount, so the edge $peter \to T$ also has flow. The combination of the flow conservation law and the added restriction that outgoing edges from variable nodes should have a flow amount of exactly 1, ensures that each variable will be assigned to exactly one value. Furthermore, the flow on edge $S \rightarrow M$ is equal to the number of occurences of value M in the solution, in this case 2, since M is assigned to $peter$ and $paul$. The flow restriction on the incoming edges of value nodes imposes the value occurence limits as defined by GCC, in this case requiring value M to appear at least once and at most twice in a solution.

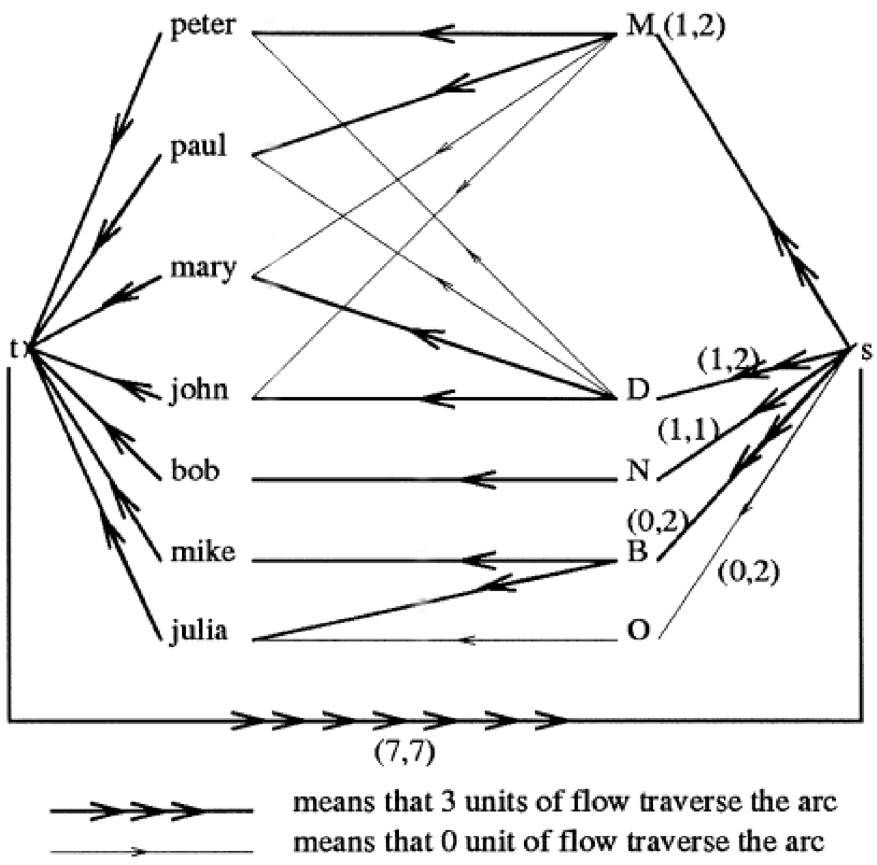
peter
paul
mary
john
bob
mike
julia
t
s
M(1,2)
D
N
B
O
(1,2)
(1,1)
(0,2)
(0,2)
(7,7)
means that 3 units of flow traverse the arc
means that 0 unit of flow traverse the arc

Figure 2: Example of a GCC Variable-Value network

In the example above we have an already established feasible flow, that corresponds to a solution. To compute this flow, we follow the method described in Chapter 2.2.2. In particular, we start with the graph shown above but without any flow on the edges. We select an edge that violates its flow restrictions, say $S \rightarrow M$ and we want to send flow to it. We work on the residual graph, which is initially identical to the variable-value network. In order to push flow into this edge, we need to find a path from M to S and send flow along it, to abide with the flow conservation law among the nodes of the graph. Therefore we push flow along the path $M \rightarrow peter \rightarrow T \rightarrow S \rightarrow M$, which "assigns" M to $peter$.

The residual graph is now updated to account for this flow change, according to the rules in Chapter 2.2.2, as follows:

- The edge $M \rightarrow peter$ is removed (since an already assigned variable-value pair cannot be assigned to itself), and a mirror edge $peter \rightarrow M$ is introduced, to allow changing the assigned value of $peter$.

- The edge $S \rightarrow M$ still exists to permit another assignment of value M to a variable, because its flow is equal to 1 but the upper limit is 2. A mirror edge $M \rightarrow S$ is not added at this point, because we cannot remove flow from this edge, since its lower flow bound is now equal to its current flow (so removing it would violate the lower bound that we just fixed). However, if in the future the flow was to be increased to 2, then we would include the mirror edge, because it would be legal to take it back to 1.
- The edge $peter \rightarrow T$ is removed, because $peter$ has been assigned to a value, and therefore we don't want to undo this operation, as we need all variables to have a value. But varible $peter$ can still swap its value with another one, and that is why the mirror edge $peter \rightarrow M$ has been included previously.

The process repeats until there are no remaining edge flow bound violations, at which point GCC is consistent.

3.3.2 Global Cardinality Constraint With Costs

For the case of costgcc, instead of simply finding a feasible flow, we need to compute a minimum cost flow with cost less than or equal to the cost upper bound. The main difference is that now we care particularly about shortest paths when sending flow, instead of any paths. If we denote m as the number of edges in $N(C)$, n the number of variables, d the number of values, and γ the greatest cost involved, then finding a min cost flow has complexity $O(nS(m, n+d, \gamma))$. We remind that S is the complexity of finding shortest paths from a node to all other nodes in a graph, as declared in Chapter 2.2.2.

Furthermore, in practice the consistency of a constraint is checked multiple times, as domains are shortened during search. In this case, there is no need to do all the work from scratch again, as we can repair the most recent flow to account for these domain changes, by using the incremental algorithm also presented in Chapter 2.2.2. The complexity in this case becomes $O(kS(m, n+d, \gamma))$, where k is the number of values that got pruned since the last run of the algorithm. This is the theoretical bound mentioned in the publication, but in our implementation which we will describe later on, k is limited to the number of values which got pruned and at the same time belonged in the most recent min cost flow, thus requiring the flow to be repaired to exclude them. Values that got pruned but didn't participate in the flow, do not result in needing to do a successive shortest paths run for them.

Let C be a consistent gcc and f be a feasible flow in $N(C)$. A value a of a variable x is not consistent with C if and only if $f_{ax} = 0$ and a and x do not belong in the same strongly connected component in $R(f)$. The intuition here is that in this case there is no cycle containing them both, which means there is no way to ever send flow through the arc (a, x). Thus, arc consistency for gcc can be achieved in $O(m + n + d)$ by computing the strongly connected components of the residual graph.

Consider $C = costgcc(X, l, u, cost, H)$ is a consistent costgcc and that f is a minimum cost flow in $N(C)$. A value a of a variable y is not consistent with C if and only if $f_{ay} = 0$ and $d_{R(f)-\{(y,a)\}}(y, a) > H - cost(f) - rc_{ay}$, where $d_{R(f)-\{(y,a)\}}(y, a)$ is the shortest path distance from y to a in the residual graph without taking into account a (y, a) edge, and rc_{ay} is the cost of (a, y) in the residual graph. The idea is that now knowing whether there is a cycle containing a particular arc is not enough. We need to find a cycle with a length greater than a given value, because costs are involved and the upper bound must be satisfied.

We care about finding shortest path distances, so arc consistency for costgcc can be achieved in $O(|\Delta|S(m, n + d, \gamma))$, where Δ is the set of values b for which $f_{sb} > 0$. While this is the main idea, some little optimizations can be applied to take advantage of the structure of the residual graph and save computations , which are explained in more detail in [2].

3.4 Implementation Details and Optimizations

Even though the publication of Régin is complete in regards to the algorithm presentation from a theoretical scope and offers some optimizations, there is still room for improvement and experimentation when we translate the algorithm to an actual program. In this section, we go through several alternative implementations that we experimented with, and we evaluate them to find the most performant one. While we do not present benchmarks for each one, we describe our findings and give a justification for the performance of them.

3.4.1 Basic Implementation

We start off with a simple implementation, which is not the most efficient one as we will show later on in this chapter, but it is a solid foundation. As soon as the constraint is posted, the value network is built, along with the residual graph. Both of these graphs are internally represented as a vector of vectors, to hold the nodes and for each node to hold its edge destination nodes, along with extra information like its lower and upper bounds and flow value. A first minimum cost flow is established by looking for lower bound violations and repairing them, sending flow along the way, and then arc consistency is applied.

As values get pruned from the variables, either due to reasoning of other constraints, or directly from branching choices of the search, costgcc is also notified and we check whether we need to verify its consistency, by utilizing Gecode's Advisors feature. The constraint is scheduled for propagation only in the case of the pruning of a variable-value pair that participates in the current min cost flow. In that case, we verify consistency by repairing the min cost flow, and then re-apply arc consistency. If the pruned variable-value did not belong in the current min cost flow, then we know for sure that the constraint is consistent, and we skip this procedure. Of course, since we don't apply arc consistency on every opportunity, we might detect some inconsistent values later on in the search tree, but this is a trade-off we are willing to make for the sake of execution time; the propagator is expensive and if we attempt to perform arc consistency on each step, we will easily observe that performance starts to deteriorate significantly.

Within the advisor, when we are notified of a domain update, Gecode provides us with the variable that got affected, but we are not given exact information about the value(s) that got pruned. Which means that to discover it, we need to iterate through the current domain of the variable and compare it with the internal state of the propagator. This is done in a fast way by holding a `varToVals` structure, which is an array of hash tables, mapping each variable to its domain. This structure is essentially the inverse of the value network, and is needed for fast lookups.

As variable-value pairs are pruned, instead of deleting the corresponding edges of the value network, we maintain them but lower their upper bounds to 0. If the upper bound of an edge which has flow becomes 0, then we need to repair the flow, as it is no longer feasible, so we schedule the propagator for execution. If not, there is no need to schedule. We push the edges that got updated in a vector called `updatedEdges`.

When the propagator is executed, we update the residual graph taking into account only the changes that happened in `updatedEdges`, and we repair the upper bound violation on them, removing flow and transferring it to other edges, according to the min cost flow algorithm. If we manage to find a new feasible min cost flow, we clear the `updatedEdges`. If at any point either the successive shortest paths method fails because there is no path that can transfer the flow, or because the total cost exceeds the upper limit, then the constraint reports failure.

Each time either a solution or failure is reported, the search tree is backtracked. To be able to revert back to previous states, Gecode copies the propagator state on each branching. In our case the graph state is copied, including the upper and lower bounds and flow, along with the residual graph. There are several helper data structures that exist throughout the lifetime of the search that do not need to be copied, like the following:

- `valToNode`: map domain values to node ids on the graph. We don't need a varToNode, because by convention we place the variables at the beginning of the nodes array, so the n^{th} variable corresponds to the n^{th} position.
- `nodeToVal`: map node ids to domain values.

Gecode does not provide a way for the propagator to choose to not copy some data structures. An initial approach to solve this is to place these structures on the heap, in the copy constructor to copy just the pointer to them, and to never delete them. This method, while it may seem efficient, is not acceptable because it suffers from a memory leak; there is no way to know when the last reference to them will become inaccessible, to finally delete them. A workaround is to upgrade to smart pointers, specifically `shared_ptr`. This way reference counting is done automatically, and there is no leaked memory.

Since the propagator is relatively expensive to run, we define its ”cost” as high, to tell to the Gecode scheduler to pass priority to cheaper propagators first, before executing costgcc. In particular we use `PropCost::cubic` with parameter `PropCost::HI`. Assume that the propagator is notified for a change in the domain of a variable x, it finds out that a value a got pruned, and inserts the edge (a, x) in `updatedEdges` and schedules the propagator. It is not necessary that it will be executed right away, as it is possible that another propagator might take precedence, and prune another value(s), say value b. In this case, costgcc will be notified again about this new change, and it will try to find which values have changed. If this change happened to be again on the same variable x as before, then it will identify both a and b as changed values, and it will insert the edge (a, x) again in `updatedEdges`. It means that it can contain duplicates, and we need to be cautious to not attempt to repair the flow or remove an edge that has already been removed. While we could use a set data structure instead of an array to ensure unique content, it would add more overhead than gain. It is cheaper to use a simple vector and just ignore an entry if we have already processed it before.

3.4.2 Improving Shortest Path Search

Regarding finding shortest paths on the residual graph for the min cost flow algorithm, Régin suggests Dijkstra’s algorithm [31], as it offers a great complexity of $O((V + E)logV)$ when implemented with a binary heap as a priority queue, where V is the number of nodes and E the number of edges in the graph.

But Dijkstra’s algorithm works only in the presence of positive costs, and the residual graph can contain negative costs. To overcome this obstacle, the costs are transformed using the **reduced costs** method, as described in [2].

If instead we use Bellman-Ford algorithm [32] which can operate with negative costs too, we do not need to maintain the reduced costs. In its original form it runs in $O(V \times E)$ time, but we can upgrade to an improvement called the **Shortest Path Faster Algorithm** [33], which although has the same complexity, in practice it can reduce the number of computations and terminate much earlier. We have found that this approach is faster than Dijkstra and reduced costs.

Note that we make this switch only for the successive shortest paths algorithm to check consistency, and not in the arc consistency. For the arc consistency, we first calculate the reduced costs on the spot by finding the shortest paths from node t to all other nodes in the residual graph using Shortest Path Faster Algorithm. This allows us to use Dijkstra’s algorithm for the shortest path searches required by the arc consistency algorithm, which proves to be efficent for several reasons. First of all, in that part of the algorithm we care about shortest paths to a specific set of nodes, and not to all. In this case Dijkstra can terminate early if it encounters all of them, as opposed to Shortest Path Faster Algorithm, which needs to do more iterations, since as soon as it finds a cost for a path to a node, there is no guarantee that it will not find a path of cheaper cost later on. Moreover, since with Dijkstra the costs are non-negative, and it always chooses the edge with the cheaper cost to advance to next, we can compare this cost with the global cost upper bound, and if we exceed it then we can terminate early, as all the edges after that point will also exceed it. These Dijkstra optimizations are mentioned in Régin's publication.

Although Régin points to a Fibonacci heap as the priority queue for Dijkstra's algorithm for slightly better complexity, we did not experiment with that, as it is common for this version to run slower in practice.

In addition, two further upgrades [34, 35] to the Shortest Path Faster Algorithm exist that can reduce the worst case number of iterations even more. Their main idea is to partition the edges into two sets and to traverse them in an order that minimizes the number of iterations. The bottleneck here is the overhead of creating and maintaining the partition, since the residual graph changes on each call to the propagator. They were attempted with two different implementation approaches, but they were proven to be significantly less performant.

3.4.3 Edge Deletion and Backtracking Without Flow

On this approach, instead of altering the upper bounds of edges to mark that we don't need them anymore, we now delete them completely from the graph. A key point of improvement here is that we also optimize the backtracking scheme for the graph, by using a technique described in [14]. According to it, each node on the graph consists of the following structure, which we will name `BtVector`:

- `list`: a vector holding its adjacent nodes, along with any additional data like edge bounds and flow value.
- `valToPos`: a hash table mapping adjacent node ids to their position in `list`.
- `listSize`: the current size of the list.

`list` and `valToPos` are backtrack stable, which means that we do not copy them on each branch, instead we always use their most recent version. The only component that needs to be backtracked is `listSize`. Initially, `list` holds all the neighbor nodes of a particular node. As an edge gets deleted, we don't remove it from `list`, but instead we swap it with the last element (also update `valToPos` to match this change), and we decrement `listSize` by one. When backtracking occurs, the old value of `listSize` is restored, and the previously deleted edges are found again at the end of the array.

The advantage of this method is that we do not need to copy the entire graph on each branch, instead we only copy one integer. Additionally, `valToPos` offers $O(1)$ element lookup and access. This method works because we only remove values, we never add new ones, except when backtracking. This data structure is also used for the `varToVals` field of the graph. As mentioned previously, this structure is the inverse of the graph, and is needed to be able to efficiently compare the propagator's internal state with the latest domain changes of a variable.

In order to not backtrack `list` and `valToPos`, we place them on the heap using `shared_ptr`, as mentioned in Chapter 3.4.1. But we have a `BtVector` structure for each node of the graph, and another for each variable node of the graph (for `varToVals`), so an identical amount of smart pointers. We notice that this becomes expensive and contributes to a noticeable performance overhead. This is caused because `shared_ptr` internally implements reference counting and has atomic operations, in order to be thread safe. We remove this overhead by taking all the content that does not need to be backtracked and wrapping it within a single smart pointer. This means that we remove the `listSize` field that belongs to `BtVector` and place it externally, so that it will be backtracked. Eventually, we end up with a `nodeListSize` array of size equal to the number of nodes, and a `varToValsSize` array of size equal to the number of variable nodes.

A crucial point of interest here is that we do not backtrack the flow. The flow values are tied to the edges in `list`, and so each time we hold the most recent flow. This has some important implications.

First of all, when the propagator is posted, we will achieve a minimum cost flow, and as values get pruned, we will also repair it if necessary and remain on a flow of minimum cost. But when we backtrack, some edges that have been previously removed will come back, which means that the most recent flow is not necessarily minimum anymore, it is possible that those fresh edges can be used to lower it further. In the case that it is no longer minimum, the residual graph will contain cycles of negative cost, meaning that if we do not take special care, the Shortest Path Faster Algorithm will be stuck in an infinite loop, always finding paths of lower and lower cost, until minus infinity.

We upgrade the Shortest Path Faster Algorithm to be able to identify negative cost loops. The way to achieve this is by counting the length of the path to each node. Without cycles, the maximum length would be equal to the number of all the nodes in the graph, so if for a node we exceed this, it means that it has to be contained within a cycle. When the propagator is executed (either after some values have been pruned or after backtracking, Gecode cannot provide this distinction to the propagator), we check for cycles. To make sure the graph is connected and that we will not miss a cycle, when the very first min cost flow is established, we include the residual edges (t, v) for each variable v, so that later we can look for cycles starting from the t node, and this way we will cover all the other nodes (by strict definition of the residual graph in chapter 2.2.2, these residual edges would normally not be included, because their flow and lower and upper bounds are all equal to 1). This inclusion does not have any side effects, as there are not any inbound nodes to t.

If a cycle is found we send flow through it, and we repeat the process, looking for more cycles and sending flow through them, until there are no more. At this point we know we have established a flow of minimum cost, so it is safe to proceed normally and iterate through `updatedEdges` and repair the flow along any edges that have been marked for deletion. A point to remember is that it is possible that the cycle repair algorithm has already removed flow from an edge which has been marked for deletion. To take this case into account, we need to check if there is still flow in a particular edge that we are processing, and if not we must skip repairing the flow for it and delete it right away.

Furthermore, we cannot backtrack the residual graph anymore. The reason for this is that it depends on the flow, and the flow is not backtracked, so if we backtracked only the residual graph then they would not be synchronized. Instead, we opt to build it from scratch every time the propagator is executed, just before we check for negative cycles. Note that we build it only during actual execution, and not at the moment that we identify that some values have been pruned and schedule the propagator, to save computational time. In addition, it is not possible to adapt the backtracking structure that we used for the graph edges, to be able to handle the residual graph too, because in the residual's case, edges can also be added, instead of only removed. The efficiency and elegance of that structure are based on the fact that it is restricted to deletions.

Finally, during the successive shortest paths, when we find a path and want to send flow to it, we need make sure to first check if the total flow cost would exceed the upper bound, and if not, to not send it and to fail. Because if we were to first change the flow and then check the cost restriction, if it was false then we would backtrack with an infeasible flow.

Although we need to do extra work at the start of each iteration to look for cycles to re-establish the optimality of the flow, we found out that this implementation is significantly faster than the base one, even when the base uses Faster Shortest Path Algorithm instead of Dijkstra's.

3.4.4 Edge Deletion and Backtracking With Flow

While the implementation described in Chapter 3.4.3 is the best so far, it raises the following question: is there a way to always land on a minimum cost flow when backtracking, so as to skip the overhead of the search for negative cost cycles each time? We answer this by experimenting with another implementation which is based on the previous one, with the modification that we now save the flow and backtrack it.

We hold a separate hash table structure which maps value nodes to sets of variable nodes, since flow in the value network will always head from value nodes to variable ones. A variable node v is included in the set of a value node u if $f_{uv} = 1$. And to know the flow value for (s, v) edges, we only need to query the size of the set that maps to value node v, this way we hold minimal information.

Even though with this choice of data structures we have $O(1)$ lookup for the flow value of an edge, it turns out that the cost of copying it during branching outweighs the benefit of skipping cycle detection, and actually runs even slower in practice.

3.4.5 Edge Deletion and Backtracking With Flow and Reduced Costs

The implementation in Chapter 3.4.3 had the property that on backtracking, the residual graph and the flow were not synchronized and we would have to rebuild the residual from scratch, since we could not backtrack it, which prohibited us from trying a reduced costs Dijkstra approach, because we would have no way to efficiently save and restore the costs. But in Chapter 3.4.4 where we now backtrack the flow, it means that we can also backtrack the residual graph and have them synchronized, and in turn the reduced costs too. However, this approach proves to be inefficient, as it is slower than all other alternatives, except for the base initial implementation of Chapter 3.4.1.

3.4.6 Residual Graph Structure

We take the fastest implementation so far, which is the one in Chapter 3.4.3, and we experiment with a more sophisticated data structure for the residual graph. Until now, it has been represented as a simple vector of vectors. For each node, we had an array that held the residual edges of it. While this is good enough for iterating the edges, searching for a particular edge and deleting an edge takes $O(\alpha)$, where α is the number of edges of a particular node. For sparse graphs this should not be a problem, but for dense ones it could be a bottleneck.

We design a structure that achieves $O(1)$ for lookup, addition, deletion and clearing, and the previous $O(\alpha)$ for iteration. The structure is a $n \times n$ matrix, where n is the number of nodes in the graph. In reality it is implemented in a single dimension for good spatial locality, but we will refer to it as two-dimensional here for simplicity.

Element at row i and column j represents an edge from node i to node j in the residual graph. We care to know if this edge actually exists, or if it has been pruned (or never existed). While we could use a boolean for that, instead we use an integer variable `active`. The residual graph maintains integer variable `activeFlag` which is a number used to identify active edges. For an edge, if `active == activeFlag` then it is active, otherwise it is not. This approach allows us to clear the graph simply by incrementing `activeFlag`, thus deactivating all edges in $O(1)$. For adding and removing edges, we make `active` equal to `activeFlag` or decrement it respectively.

While the search and retrieval of a particular edge takes $O(1)$, we would also like to be able to iterate all the neighbors of a particular node, without taking extra steps. For each row i, we hold `start` and `end`, which indicate the array position of the first and last outgoing edges from i. When adding an edge, we go directly to its last edge, and add it at the end. Each edge for row i also includes `next` and `prev` variables, which point to the next and previous edge of it respectively. This way all the edges are linked and we can iterate them without taking extra steps, and we also know the start and end of this chain list. During addition and deletion, all these fields are updated accordingly to maintain the chain.

Evaluating this structure through experiments, we find out that it is of neutral impact on sparse graphs. On an instance of 100 variables and 100 values, resulting in a complete graph of 10000 nodes, we start to see a 7% improvement in the number of solutions reported over a time limit of 5 minutes. Increasing the variables to 1000, there is a 5% deterioration. Increasing both the variables and values to 1000, thus having a graph of 1 million nodes, we notice a 41% decrease in the number of solutions. In this instance, memory consumption is 335MB compared to 260MB with simple vectors.

Even though on first glance this approach can seem to benefit us when running on dense graphs, the graph can easily grow too large, as the memory consumption is $O(n^2)$. We speculate that performance starts to drop because of the unreasonable memory usage, resulting in more page faults.

We can improve the memory consumption and tighten the array further. We know for sure that there are some bounds for specific edges, for example:

- S node can only point to value nodes.
- T node can only point to variable nodes and S.
- Variable nodes can only point to value nodes **in their domain** and to T node.
- Value nodes can only point to variable nodes **if the value belongs to the respective variable domain**.

Taking these into account, we do not need to hold the whole $n \times n$ array. With some extra logic, we can carefully minimize the amount of useless stored information. However, we did not attempt this memory optimization, as we thought that the potential gain did not seem to be strong enough to justify the development time. In fact, eventually we opted in to use just the initial vector of vectors implementation, for the sake of simplicity, but also better performance on large graphs.

3.5 Custom Branching Heuristic

In the practical improvements section of the original paper by Régin, the use of the **Max Regret** branching heuristic is suggested. The regret of a variable is defined as the cost distance between its best and second best assignments. The variable with the maximum regret is chosen. The idea is that if we do not choose this variable and if this variable is instantiated with a value different from the one leading to the best assignment, we will have to pay at least the value of the regret.

Because in the arc consistency algorithm we calculate shortest paths from each variable to every other possible value in its domain, we are able to calculate the precise regret values, as opposed to the common case in other problems, where we simply approximate them, thus the reason why this heuristic was proposed. However, it is not practical. Assume that we branch on a variable of maximum regret. This branching might cause the regret values of the other variables to change, but because our propagator is not necessarily executed right away on a domain change, we cannot update the regret values, because they are updated as part of the arc consistency algorithm. So the next choices based on max regret will not necessarily be correct. We also experimented with ordering our propagator to always perform arc consistency, regardless of if we know that it's consistent, in order to check if the max regret heuristic could provide a good enough gain to outweigh the overhead induced by this modification, but it was not the case.

While the Max Regret heuristic described by Régin concerns choosing which variable to branch on next, instead we propose and implement a different one that determines which **value** to choose. Our heuristic is lightweight, does not depend on arc consistency, and it proves to dramatically reduce the number of failed search nodes, and for most cases, to improve running time by a large factor.

The costgcc finds all feasible solutions that satisfy the capacity and demand restrictions for the values, that have a cost lower or equal to a given bound. However, one important property of the algorithm is that internally it will always compute a min cost flow, meaning that every time it checks for feasibility, it will find an assignment of minimum cost. We can forward this information to the brancher and use it as a heuristic, to prioritize branching on values that are already known to form a solution.

Since this is an interesting property, for the remainder of this chapter we make an observation about the order that solutions are reported when it comes to their optimality by using this heuristic, and we make a distinction between when the constraint is used alone, and when there are more constraints interfering on the same variable set.

3.5.1 Order of solutions: costgcc on its own

When the constraint is first posted, we naturally compute a min cost flow to check for feasibility. During the search, we branch according to the heuristic, so we go directly to a solution of optimal cost. After we report it, when we branch on an alternative choice and thus we prune a value, the constraint is re-checked for feasibility, and so another min cost flow is computed. Our heuristic is updated with the new optimal solution. The search subtree below that alternative will therefore lead straight to another optimal solution.

As we traverse down a search tree we are removing values, so the costs of solutions found down a subtree will be equal or higher to the parent ones. When we return to the root node to try an alternative choice and thus form a different subtree, the same will occur. However, we do not have any relation between the costs of two sibling subtrees, which means that if we look for multiple solutions using this branching strategy, they will not necessarily appear with same or increasing cost compared to the costs of solutions reported previously from a sibling subtree. This is because while we do make sure to choose the values to branch on for each variable to lead to a solution of optimal cost, we do not have any logic for prioritizing which variables to choose first for branching.

In the case of costgcc alone, the first solution that is reported is guaranteed to be of optimal cost, which means that if we use costgcc with our proposed branching strategy and ask Gecode for exactly one solution, we will receive one with the lowest possible cost, and not just one with cost lower than the upper bound provided.

3.5.2 Order of solutions: costgcc in conjunction with other constraints

Assume that we have found a min cost flow and search has assigned some variables to the respective values that follow this min cost flow, but not all yet. If at this moment another constraint causes the pruning of a value which is used by our min cost flow (and we haven't branched on it yet), then the min cost flow will change, to account for the removal of that value, and so the values on which we will branch next will also be updated. But in this case, the new min cost flow will be optimal only for the subproblem in which some variables have already been assigned to some specific values, but not necessarily for the original problem. So it is possible that the pruning of a value from an external constraint will lead us first to a more costly solution than the optimal one. Nevertheless, of course if we look for all solutions and not just one, we will receive them all, and the branching heuristic will still minimize failed search nodes and be effective.

3.6 Interest in Optimization Problems

In the previous section we described a branching strategy that not only drastically improves running time, but if used without other constraints, it can also provide us with a solution of optimal cost, instead of just a solution of cost lower than the bound given, which was the original purpose of this constraint. This restricted use case of the costgcc alone is not of much interest, as in a real world scenario, usually multiple different constraints are combined together to solve complex problems. This raises the question: is there a way to extend costgcc's applications, by enabling it to efficiently solve optimization problems combined with other constraints, in which the goal is to minimize a global cost function?

Note that it is not essential to use our custom branching strategy anymore, but it is still highly recommended to improve performance. We can start by using a branch and bound search method. Each time the search finds a solution, it will try to improve it by looking for one with smaller cost. This method will work also in conjunction with other constraints too, as all solutions will be tested, and the ones with suboptimal cost than the current best one will be rejected.

While this method is correct, it is not the best we can do. We can improve it by taking advantage of the upper bound limit for the cost of costgcc. Internally, the algorithm computes a min cost flow, and if anytime it finds out that its cost is higher than the bound, it reports failure. In addition, this limit can be used to further prune values during arc consistency. Since branch and bound tries to find solutions with a decreasing upper cost bound each time, we can adapt this bound change internally to costgcc as well. We distinguish between two cases:

- **Case A**: The cost variable that we wish to minimize depends only on costgcc.
- **Case B**: The cost variable that we wish to minimize depends on costgcc and external constraints.

In both cases, multiple constraints can co-exist with costgcc, but the important part is how the cost variable is constrained. For case A, the cost upper bound of costgcc can always be identical to the cost bound used for the branch and bound method, and be updated as we find solutions of lower cost, since it doesn't depend on anything else. For case B, the cost upper bound for costgcc must be large enough to cover the worst case scenario of the external constraints. It would be incorrect to use the min cost flow of the most recent solution as the new bound for the branch and bound, because it is possible that a better solution could exist that has higher min cost flow than the most recent one, but lower cost overall because of the cost participation of the external constraints.

Some examples for each case respectively are the **Travelling Saleman Problem** and the **Warehouse Location Problem**, which are explained in more detail in Chapter 5. Generally, case A results in much better performance than case B, since it allows us to strongly restrict the upper bound of costgcc. The obligation to comply with the worst case scenario of the external constraints of case B can result in large upper bounds, which certainly do not provide efficient propagation.

4. SYMMETRIC GLOBAL CARDINALITY

The symmetric global cardinality constraint (symgcc) is similar to the original gcc, but defined on set variables instead of integer variables. That is, variables whose domain is a set of sets of possible values. Aside from the usual lower and upper bound restrictions on the occurrences of the values, there exist additional constraints on the set cardinality of each variable. More specifically, each variable has a lower and an upper bound for its set cardinality.

Similarly to the gcc, this constraint can arise in many scheduling problems. Consider a simple example in which there are workers and tasks to be executed. Each worker X_i is capable of executing a specific set of tasks, and must take responsibility for at least l_{x_i} tasks and at most u_{x_i}. Moreover, each task j requires at least l_{v_j} people to be dedicated to it, and at most u_{v_j}. This can be modeled with symgcc.

More formally, for a given assignment P, let $P(x_i)$ denote the value assigned to the variable x_i by P and $\#(x_i, P)$, the cardinality $|P(x_i)|$ of the set $P(x_i)$ and for any constraint C and element $v_j \in D(C), \#(v_j, C, P)$ denote the number of occurrences of v_j in the values assigned by P to the variables . If $T(C)$ is a subset of the Cartesian product of the domain of each variable that specifies the allowed combinations of values for the variables, then a symmetric cardinality constraint is a constraint C over a set of variables $X(C)$ which associates with each variable $x_i \in X(C)$ two non-negative integers l_{x_i} and u_{x_i}, and with each value $v_j \in D(C)$ two other non-negative integers l_{v_j} and u_{v_j}, such that a restriction of an assignment P to the variables in $X(C)$ is an element in $T(C)$ iff

$$\forall i(l_{x_i} \leq \#(x_i, P) \leq u_{x_i}) \text{ and } \forall j(l_{v_j} \leq \#(v_j, C, P) \leq u_{v_j})$$

The code developed for this constraint, along with a usage example for the experiments conducted on Chapter 5.2.1 can be found in the Github repository in [30] under the branch name *symgcc*, in the subdirectory *gecode/set/gcc*.

4.1 Modeling With Gecode

As there is no gcc defined on set variables in Gecode, one can achieve symgcc with the model described below. Consider $X = \{X_1, ..., X_n\}$ to be the set of variables involved in symgcc, $D = \{D_1, ..., D_m\}$ to be the set of different possible values, and $l_{x_i}, u_{x_i}, l_{v_i}, u_{v_i}$ as the lower and upper bounds of the variable cardinalities and values respectively.

1. Define B as a boolean variable array of n rows and m columns. Assigning B_{ij} to *true* means that value D_j is included in variable the X_i set, while assigning to *false* means that the value is excluded from the set. Set B_{ij} to *false* for each value D_j that does not belong to the domain of variable X_i, either because it got pruned or because it never belonged in the first place.

2. Constrain the sum of the variables of each column j to be greater than or equal to l_{v_j} and less than or equal to u_{v_j}.
3. Constrain the sum of the variables of each row i to be greater than or equal to l_{x_i} and less than or equal to u_{x_i}.

Steps 2 and 3 can be achieved with the use of `linear` or `count` constraints, since boolean values are represented with the numbers 0 and 1. This decomposition can miss some pruning during arc consistency, because it cannot examine the whole picture of the problem globally, hence the motivation for implementing a dedicated symgcc.

4.2 Constraint Usage

The symgcc we have implemented is

```
countSet(home, X, D, LVAL, UVAL, LVAR, UVAR, P)
```

as follows:

- **home**: the current Gecode home space. Mandatory argument for all Gecode constraints.
- **X**: array of the variables to constrain (type `SetVarArgs`).
- **D**: array of all the different possible domain values (type `IntArgs`).
- **LVAL**: array containing the lower bound for each value in D (type `IntArgs`).
- **UVAL**: array containing the upper bound for each value in D (type `IntArgs`).
- **LVAR**: array containing the lower bound for each variable in X (type `IntArgs`).
- **UVAR**: array containing the upper bound for each variable in X (type `IntArgs`).
- **P**: optional argument. Specifies propagation level, accepts `IPL_DOM` for arc consistency or `IPL_VAL` for just checking if the constraint holds, but without pruning any values. Optional argument, default value is `IPL_DOM`.

An exception is thrown if one of the following conditions involved around the arguments is not met:

- X should not contain duplicate variables and should be of at least size 1.
- D should not contain duplicates.
- D should include all the values from the domains of the variables in X.
- $LVAL$ and $UVAL$ arrays should be the same size as D.
- $LVAR$ and $UVAR$ arrays should be the same size as X.
- Bounds in $LVAL$, $UVAL$, $LVAR$ and $UVAR$ should be non-negative.
- Each lower bound $LVAL[i]$ should be smaller or equal to the respective upper bound

$UVAL[i]$.

- Each lower bound $LVAR[i]$ should be smaller or equal to the respective upper bound $UVAR[i]$.

4.3 Algorithm

The symgcc is a natural extension of the original gcc. We will consider again the way we approach the gcc in Chapter 3.3. The value network of symgcc is the same, with the following two changes:

- For the edges from each variable x to node t, the lower and upper bounds follow the cardinality constraints for said variable.
- The lower and upper bound of the edge from t to s are zero and infinite respectively.

The new bounds on the edge from t to s describe that it is not necessary for a variable to take a value, as it is legal to have variables be assigned to an empty set (as long as their cardinality lower bound is zero).

Aside from this change, the algorithm for symgcc is identical to gcc. We need to find a feasible flow to prove consistency, and for arc consistency we care about the strongly connected components in the residual graph. A flow that is feasible means that all the edge bound restrictions will be satisfied, and thus the cardinality and value constraints will be met. Regarding arc consistency, a variable and a value node which have no flow going through their edge, and that belong in a different strongly connected component, will never be able to be assigned to each other, and thus can be pruned. There is no cycle involving them, and thus we will never be able to send flow through their edge. Arc consistency can be achieved in $O(m+n+d)$ where m is the number of edges, n is the number of variables and d is the number of values.

The incremental aspect of the algorithm is also identical to the original gcc, as during each iteration of value pruning we can repair the flow based on the previous one and without needing to compute it from scratch.

Compared to costgcc, symgcc is simpler and easier to implement, because of the absence of costs. It means that for finding shortest paths, we can just use DFS or BFS instead of complicated cost-based shortest path algorithms, and for arc consistency, we care about just existence of nodes in strongly connected components, without needing to take into account their path cost in addition.

4.4 Implementation Details and Optimizations

The original publication for symgcc does not mention any implementation details or optimizations. In this section, we will present an initial naive approach, and then a way to speed up the algorithm in practice. Because of the similarity for the data structures used between this constraint and costgcc, we can reuse our research findings and utilize the best implementation ideas, without needing to re-invent the wheel.

4.4.1 Basic Implementation

The base implementation is the same as the one described in Chapter 3.4.1, for costgcc. Therefore, we will only mention any differences. The main one is that since there are no costs involved, we care about finding a feasible flow and not a min cost flow.

In Gecode, since the domain of a set variable can be exponentially large, they are described by using 2 set intervals: the Greatest Lower Bound (GLB) and the Least Upper Bound (LUB). GLB includes all values that are for sure *known* to be included in the set, while LUB includes values that *could* be included . However, these two interval bounds cannot fully represent the domain of a set. Consider the example domain of $\Big\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\}\Big\}$. This domain cannot be captured exactly by an interval, as the closest one would be a GLB of $\{\}$ and an LUB of $\{1,2,3\}$. Therefore, in addition Gecode adopts min and max cardinalities for this purpose. Thus, the domain described from the above intervals would be further restricted by stating that the min cardinality should be 1 and the max cardinality should be 2.

When dealing with set variables, value prunings are described by changes in the interval sets or the cardinality bounds. Including a value in the GLB implies that all other sets of cardinality equal to the one of GLB that are different from GLB, cannot be assigned to the variable. Excluding a value from LUB implies that it will never be included in the set. Increasing the min cardinality means that all sets of lower cardinality will be pruned from the domain, while decreasing the max cardinality means that all sets of greater cardinality will be excluded.

Therefore, to check what values got pruned during each iteration, we iterate the GLB and LUB sets, and in addition we check the min and max cardinalities. If a value is included in the GLB of a variable but has no flow in the corresponding edge of the graph, we change its lower bound from 0 to 1 and report a lower bound violation. If the edge corresponding to a variable-value has an upper bound of 1 but does not exist in the LUB, we reduce the upper bound to 0 to mark that it is pruned, and if it also had flow, we mark it as an upper bound violation. We update the bounds on the edges from a variable to t node to match the cardinality bounds for said variable. If the flow is no longer between the bounds due to this change, we report another bound violation. All these violating edges are included in a vector, and they are repaired by the incremental algorithm at the next execution of the propagator.

A violation in the bounds of a value-variable edge will be repaired in one step by the incremental algorithm, when the violating edge is met and examined. However, a change in cardinalities could require more steps. If the min or max cardinalities are changed by more than 1, then we need to take care to remain on this violation and keep repairing until the flow becomes within the bounds. For instance, if the min cardinality of a variable x turns to 2 from 0 and the flow through the corresponding edge (x, t) was 0, we need to remember to consider it twice, in order to achieve at least the lower bound of 2, because flow is added or removed from the network at a value of 1 each time.

To find the strongly connected components, we implement an iterative version of Tarjan's algorithm [12]. The iterative approach is much more efficient than a recursive one, as it limits expensive function calls to the stack. The algorithm requires certain data structures of size equal to the number of nodes in the graph, which need to be cleared at the beginning of each run. These structures state which SCC id each node belongs to, and also mark if a node has been visited during the algorithm execution. To avoid constantly clearing them, we allocate them only once and use a *certificate* integer number to mark valid elements between two iterations. At the end of each run, the certificate is updated to be larger than the maximum element in the structures, instantly invalidating all the elements in $O(1)$. We compare against this certificate within the algorithm, to know about element validity and existence. SCC ids that are lower than the certificate are considered as invalid.

4.4.2 Edge Deletion and Backtracking Without Flow

This idea is taken from the survey in [14], and is the exact same one as described in Chapter 3.4.3. Essentially, instead of altering the lower and upper bounds on the edges of the graph, we delete edges that correspond to pruned values. Additionally, the graph is backtracked in a space efficient way, minimizing unnecessary overhead. Since we don't change bounds on the edges, we take a note in an array of which edges are violating them each time, and we repair them on the next execution, clearing the array after. When building the residual graph, normally we would compare an edge's flow against its lower and upper bounds, but since these bounds are not altered in this backtrack-efficient implementation, instead we compare directly with a variable's GLB, LUB and min and max cardinalities, to obtain the ”implied” edge bounds.

4.4.3 Important Edges

An optimization mentioned in [14] for the extended global cardinality constraint (EGCC) deals with *important edges*. The idea is that the edges of the graph can be partitioned into *important* and *unimportant*. Only the the removal of important ones can result in further pruning, and so the propagator can skip execution when unimportant edges are removed. The important edges are decided from within Tarjan's algorithm to find the strongly connected components. Only minor instrumentation in the code is required to maintain them, but their bottleneck is the fact that they need to be backtracked, thus copied during each cloning of the propagator. They were attempted, but they either provided no significant gain or made performance worse because of the copying overhead.

4.4.4 Dynamic Partitioning

Another optimization for EGCC found in [14] is Dynamic Partitioning. That is, the realization that after some value prunings, symgcc could be split into multiple independent constraints. Essentially, this would allow the propagator to deal with a smaller, more focused graph each time internally, depending on which partition of the constraint is triggered based on the values that got pruned during each iteration. The benefit would come in Tarjan's algorithm for the search of the strongly connected components, as the search would contain itself within a subset of the graph, ignoring unnecessary computations.

This optimization was not implemented, however it is still mentioned as it could be a point of future experimentation. Since symgcc and gcc are very closely related to each other, it is likely that it could prove to be beneficial for our case too.

4.5 Custom Branching Heuristic

In Chapter 3.5, we described a powerful branching heuristic that is capable of dramatically reducing runtime. It is based on the property that during each consistency check of the propagator, a feasible flow is calculated internally, which represents a legal solution to the problem. This solution can be used to guide the search branching, and be updated on the go as we deal with alternative choices.

While at first look it could seem like we could do this on symgcc too, unfortunately it is not possible. This is due to Gecode's way to branch on set variables. According to it, we can either include values in the GLB, exclude from the LUB, or tighten the cardinalities. For our heuristic to work, we would need to be able to branch by strict equality and inequality. That is, to say that a variable should be equal to an exact value set, and to remove said value set as the alternative choice. We are able to achieve the equality part by including the necessary elements in the GLB and reducing the max cardinality to be equal to the cardinality of the GLB. But we cannot do something similar for the inequality part. If we remove a value from LUB or tighten the max cardinality, we are already potentially removing more set values from the domain than we would like.

One way to work around this restriction would be to use an array of boolean variables, to represent if a value is included on a set variable or not, which is the default way to model symgcc in Gecode, as described in Chapter 4.1. This would allow the use of the custom branching heuristic, as we would be able to easily branch on equality and inequality on boolean variables instead of set ones. But we speculate that it would not help in practice, as it would drastically increase the number of variables required to represent our initial problem, requiring a variable for each possible value for each set variable.

5. EXPERIMENTAL EVALUATION

In this section, we conduct experiments to evaluate the performance of our programs. We care to observe the behaviour of their various propagation levels, to discover patterns to know which configuration is best to use for each case, and to compare to the alternatives that Gecode offers. For the remaining of this chapter, we will use the following terms to refer to different configurations for our programs:

- **Val**: no domain pruning
- **Val B**: no domain pruning, use our custom branching heuristic
- **Dom**: arc consistency
- **Dom B**: arc consistency, use our custom branching heuristic

The custom branching heuristic configurations appear only on costgcc, as we have explained in Chapter 4.5 that is not possible to use it for symgcc as well. We will also adopt the term **Multi** to refer to using a model that decomposes costgcc / symgcc to multiple constraints, to achieve the same behaviour . We will use it to compare the performance of our constraints to using constraints that already come with Gecode.

All experiments were performed on a 64-bit Intel® CoreTM i7-7500U CPU @ 2.70GHz / 3.50Ghz Turbo $\times$ 4 machine with 16GB memory, running Ubuntu 18.04.1 LTS and Gecode version 6.3.1.

5.1 Global Cardinality With Costs

For costgcc, we will choose the implementation described in Chapter 3.4.3, as we have found that it is the best one.

The custom branching heuristic is the one mentioned in Chapter 3.5. It's a method for choosing which value to branch on next. For choosing the variable, we will use the **Max Regret** heuristic, as described on that same chapter. The regret value for each variable is not exact, but is approximated as the difference between the lowest and second lowest costs involving that variable. We have found that it is generally more effective than a simple MRV. For the cases when we don't use our custom branching heuristic, we will branch on values by simply selecting the minimum one.

Since we were unable to find real world problem data that covers all the aspects of costgcc at once, we will benchmark this part by generating random instances. Next, we will study its behaviour on optimization problems, first on the **Traveling Salesman Problem**, and after briefly on the **Warehouse Location Problem**. For these two real world problems, we will use input data taken from benchmark libraries.

5.1.1 Randomly generated instances

We generate data using a separate program, according to the following parameters. All of them are followed by space and then a number, except for `-f` , which is followed by string that specifies the name of the output file.

- `-n`: number of variables
- `-m`: number of values
- `-p`: percentage representing the density of variable-value graph
- `-umin`: minimum upper bounds percentage
- `-umax`: maximum upper bounds percentage
- `-lmin`: minimum lower bounds percentage
- `-lmax`: maximum lower bounds percentage
- `-cmin`: minimum possible cost value
- `-cmax`: maximum possible cost value
- `-c`: cost upper bound
- `-s`: seed for random generator

The upper bound percentages are in relation to the total occurrences number of a specific value in the domains of the variables, while the lower bound percentages are in relation to the upper bounds of each value. If the seed is not specified, then it will be automatically generated and printed, so that identical data can be replicated in the future.

We create the following files with the respective commands, that we will use for experimentation in this chapter:

- big: `./gendata -f big -n 100 -m 100 -p 100 -umin 20 -umax 30 -lmin 0 -lmax 8 -c 100 -s 294367347`
- big-sparse: `./gendata -f bigSparse -n 100 -m 100 -p 10 -umin 20 -umax 30 -lmin 0 -lmax 50 -c 171 -s 294367347`
- big-var-low-val: `./gendata -f bigVarLowVal -n 100 -m 25 -p 100 -umin 20 -umax 30 -lmin 0 -lmax 8 -c 109 -s 294367347`
- big-var-low-val-sparse: `./gendata -f bigVarLowValSparse -n 100 -m 25 -p 10 -umin 50 -umax 75 -lmin 0 -lmax 0 -c 386 -s 294367347`
- low-var-big-val: `./gendata -f lowVarBigVal -n 25 -m 100 -p 100 -umin 20 -umax 30 -lmin 0 -lmax 8 -c 25 -s 294367347`
- low-var-big-val-sparse: `./gendata -f lowVarBigValSparse -n 25 -m 100 -p 10 -umin 20 -umax 30 -lmin 0 -lmax 50 -c 39 -s 294367347`

We will compare our program's different configurations with themselves, and with a model that substitutes costgcc by using a conjunction of gcc and sum. In Gecode, these can be achieved with `count` and `linear` respectively, and we will refer to this decomposition by the term *Multi*. In the results that we will showcase, for Multi we use only the best configuration for each cost upper bound case, between no pruning, arc consistency, and bounds consistency. For instances *big*, *bigSparse* and *bigVarLowVal*, arc consistency is the best choice for Multi, and for instances *bigVarLowValSparse*, *lowVarBigVal* and *lowVarBigValSparse*, no pruning at all is better.

Internally, as of Gecode version 6.3.1, gcc bound consistency is based on [16], and domain consistency is based on [13] (even though in the latest documentation version which is 6.2.0, it is stated that it follows Régin's [10] approach).

The interest of using costgcc compared to a simple gcc is when the cost sum is constrained. For each data instance, we start to do our comparisons by setting the cost upper bound to be equal to the minimum possible cost sum of a valid solution (we will call it *min cost*). We then increase it gradually, until we reach the point where we notice that Multi performs better than our costgcc configurations. In order to find the min cost, we only make a modification in the code so that costgcc will print the cost of the first min cost flow it will find.

We run our program with a timeout of 1 minute, and report the number of solutions found within this limit. We begin with instance *big*, which consists of 100 variables and 100 values with full density, forming a complete variable-value graph. From Figure 3 and Table 1, we can make the following observations:

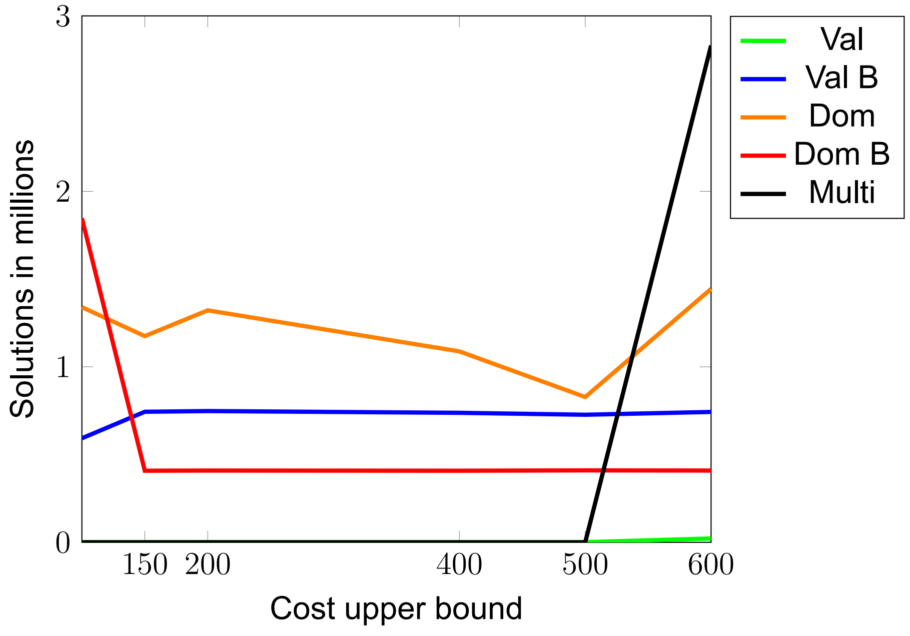

Figure 3: Number of solutions for increasingly higher cost upper bound for instance *big*

Table 1: Number of failures for different configurations and cost upper bounds for instance *big*

	Val	$ValB$	Dom	$DomB$
100	$2036K$	$1117K$	$1638K$	$41K$
150	$1940K$	$918K$	$1542K$	$119K$
200	$1837K$	$922K$	$1308K$	$119K$
400	$2200K$	$911K$	$1324K$	$119K$
500	$3027K$	$898K$	$1262K$	$119K$
600	$4013K$	$917K$	$718K$	$119K$

- Dom B is the best only for when the cost upper bound is equal to the minimum cost. For the other cases it remains consistent, but Dom and Val B perform better.
- Val B is far superior to Val, which is the worst.
- Multi is always failing while finding no solutions at all, except for when the cost upper bound is loosened enough, when it outperforms every other configuration by a large factor. At this point, the cost sum is unrestricted enough that the costs barely matter.
- Dom B always has the least amount of failures.

We now experiment on a sparse graph of the same dimensions, in *bigSparse*. In Figure 4 and Table 2 we observe:

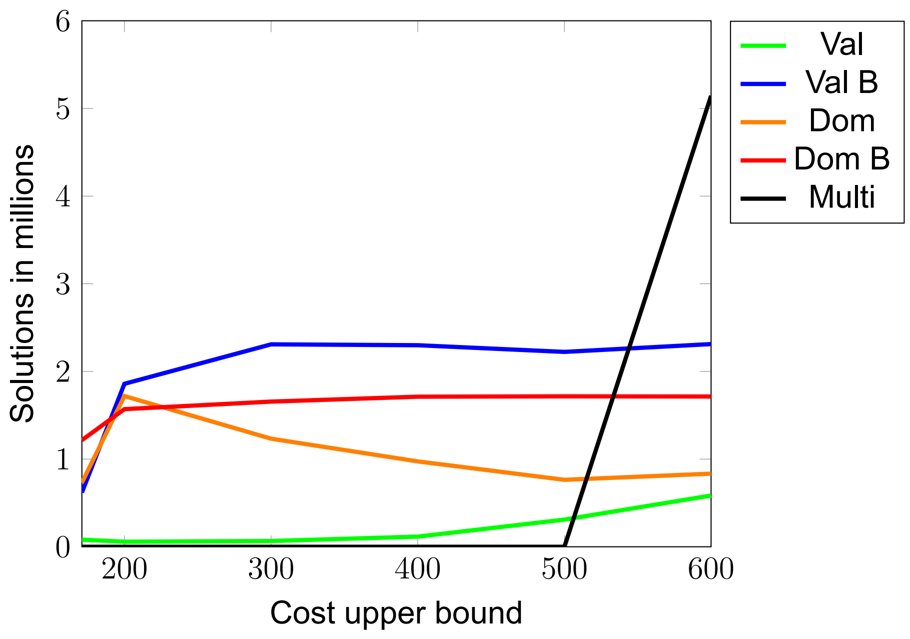

Figure 4: Number of solutions for increasingly higher cost upper bound for instance *big-sparse*

Table 2: Number of failures for different configurations and cost upper bounds for instance *bigSparse*

	Val	$ValB$	Dom	$DomB$
171	$3486K$	$2575K$	0	$16K$
200	$3484K$	$916K$	4	$7K$
300	$3673K$	$834K$	$16K$	$4K$
400	$3763K$	$830K$	466	$4K$
500	$4471K$	$803K$	$11K$	$4K$
600	$2460K$	$834K$	58	$4K$

- Dom B is again the best choice for when the cost bound is equal to the minimum cost.
- The best configuration overall is Val B.
- Even though Dom B has more failures than Dom, it still performs better for most cases.
- Val B dramatically reduces the number of failures from Val.
- Multi finds no solutions, until the cost sum is loosened enough when it outperforms every other configuration.

Further on, we keep the number of variables at 100 but decrease the total values to 25, resulting in a complete variable-value graph found in *bigVarLowVal*. From Figure 5 and Table 3, we can see the following:

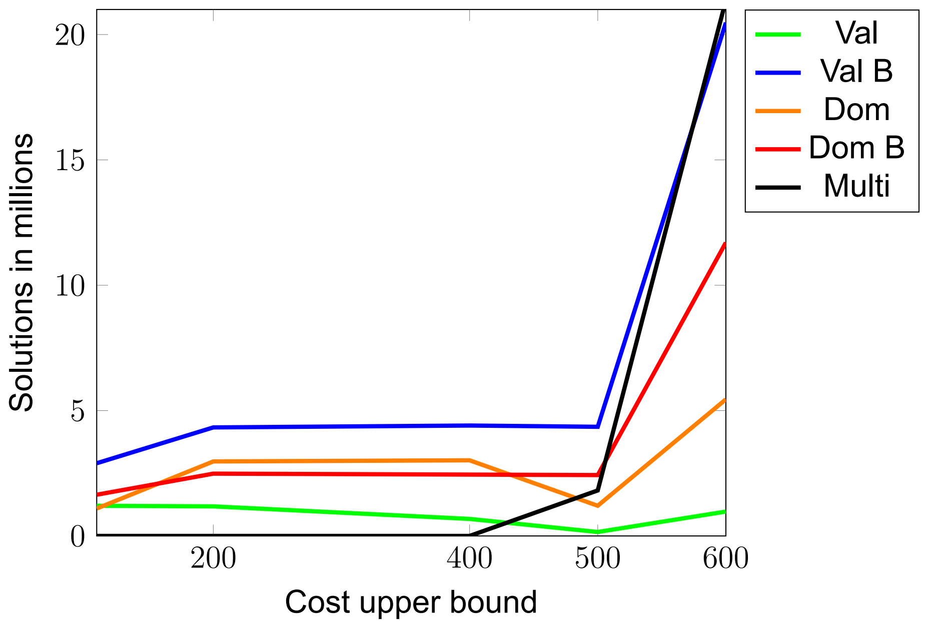

Figure 5: Number of solutions for increasingly higher cost upper bound for instance *big-var-low-val*

Table 3: Number of failures for different configurations and cost upper bounds, for instance *big-var-low-val*

	Val	$ValB$	Dom	$DomB$
109	$4218K$	$1736K$	700	$27K$
200	$4148K$	$740K$	4	$9K$
400	$4662K$	$758K$	$11K$	$9K$
500	$6425K$	$748K$	$1235K$	$8K$
600	$30945K$	$3710K$	$8468K$	$50K$

- Val B is always superior to every other configuration.
- Dom B seems to be the better choice compared to Dom, even though the latter performs better for some cost bounds.
- Using the custom branching on Dom sometimes reduces failures, while other times introduces more.
- Contrary to the previous datasets, after we start finding solutions for Multi, costgcc is still better for a while.

We take the previous instance dimensions, but make the variable-value graph more sparse, in *bigVarLowValSparse*. From Figure 6 and Table 4, we observe:

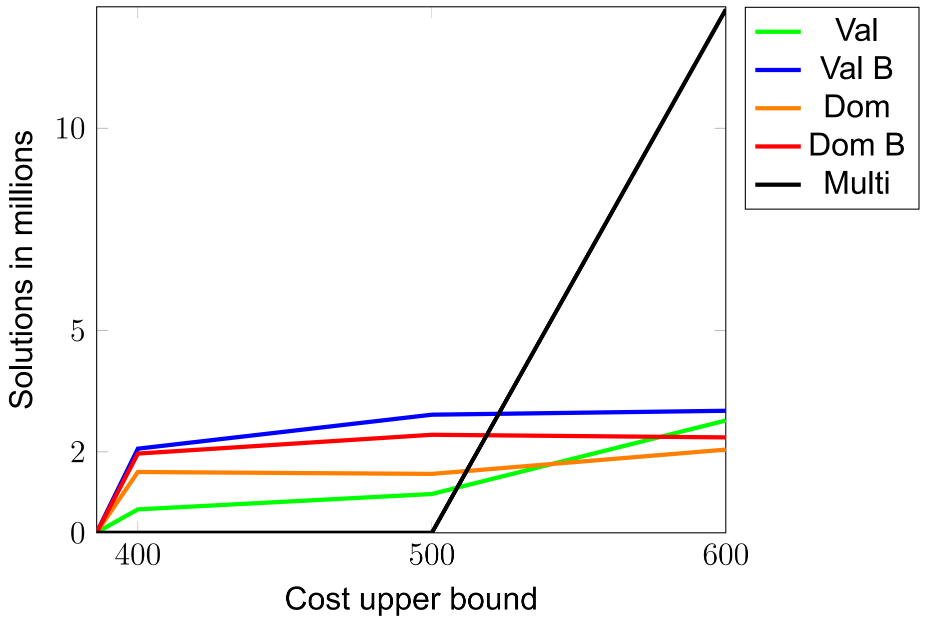

Figure 6: Number of solutions for increasingly higher cost upper bound for instance *big-var-low-val-sparse*

Table 4: Number of failures for different configurations and cost upper bounds, for instance *big-var-low-val-sparse*

	Val	$ValB$	Dom	$DomB$
386	$126K$	$69K$	0	0
400	$5094K$	$2218K$	568	$45K$
500	$4358K$	$516K$	$6K$	220
600	$178K$	$527K$	0	215

- When the cost upper bound is equal to the minimum cost, all costgcc configurations find all the solutions instantly in less than a second, with only Val taking 1.5 seconds, while Multi does not find a single solution within the 1 minute cutoff.
- Val B is the best one.
- Dom B is better than Dom.
- On the highest cost upper bound, Val manages to overcome Dom and Dom B. This is because the cost sum becomes unrestricted enough that the overhead of Dom does not justify the (lack of) pruning benefit.

We experiment with reducing the variables to 25, while keeping the values at 100. In Figure 7 and Table 5 we observe:

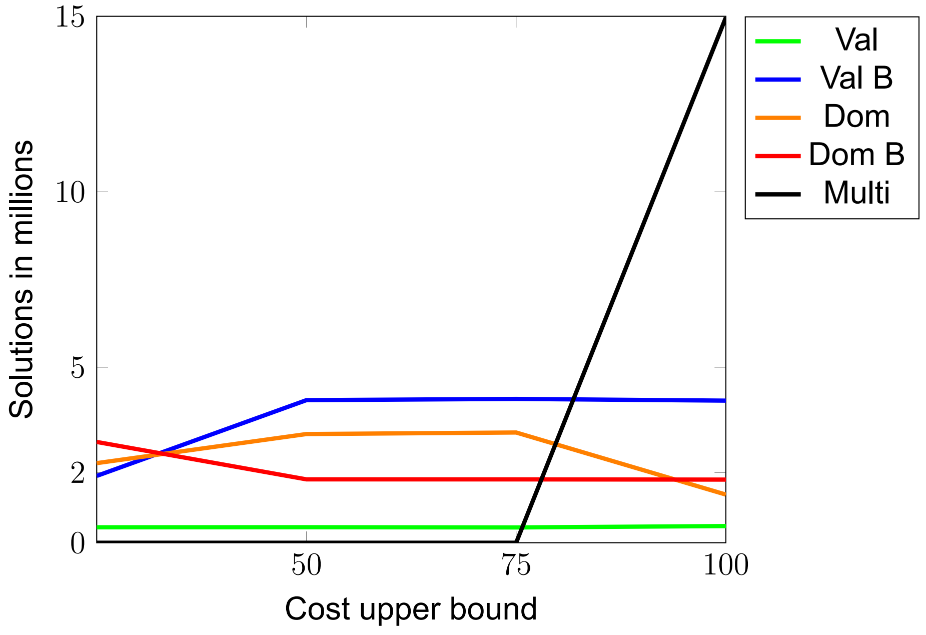

Figure 7: Number of solutions for increasingly higher cost upper bound for instance *low-var-big-val*

Table 5: Number of failures for different configurations and cost upper bounds, for instance *low-var-big-val*

	Val	$ValB$	Dom	$DomB$
25	$4721K$	$230K$	0	0
50	$4735K$	$1K$	0	0
75	$4675K$	6	0	0
100	$4155K$	6	0	0

- Dom B is best only for the minimum cost upper bound.
- Dom and Dom B reduce failures to none for all cost upper bounds. Val B also has only a few.
- Val B is the best overall.

Finally, we take the previous instance and make it sparse. In Figure 8 and Table 6 , we can see:

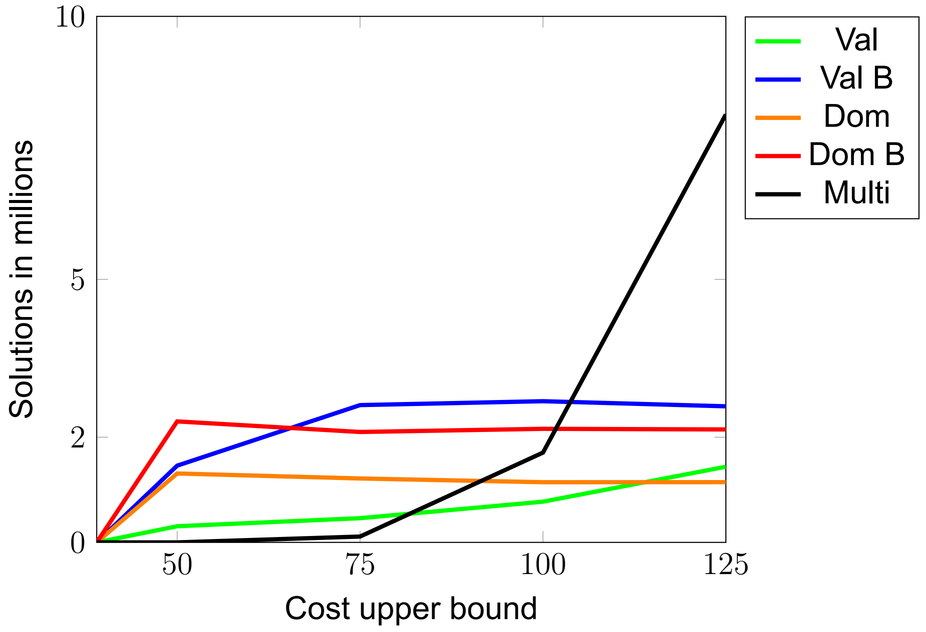

Figure 8: Number of solutions for increasingly higher cost upper bound for instance *low-var-big-val-sparse*

Table 6: Number of failures for different configurations and cost upper bounds, for instance *low-var-big-val-sparse*

	Val	$ValB$	Dom	$DomB$
39	$3K$	740	1	0
50	$5524K$	$2570K$	$27K$	$10K$
75	$4663K$	$1137K$	$2K$	0
100	$3380K$	$1084K$	158	0
125	$516K$	$1045K$	33	0

- Costgcc wins for a while even after multi starts to find solutions.
- When the cost upper bound is the minimum, costgcc finds all solutions instantly with all configurations.
- Dom and Dom B have limited failures.
- Dom B is better than Dom.
- Val B is the best overall except for when the cost upper bound is low.
- For the highest cost upper bound, Val manages to beat Dom, because the sum is too unrestricted and the overhead of Dom is not worth it, as it cannot prune many values.

Across all these different instances, we gather our conclusions:

- The best configuration overall is Val B. While Val is not promising, if we upgrade it with our custom branching heuristic, it transforms into a powerful method, as it is lightweight yet effective.
- For most instances, Multi is stuck in a failure loop without finding a single solution, but when it finally escapes, it outperforms every other configuration by a large factor. This is because at that point, the cost upper bound is so loose that the costs are not important anymore. Gcc does not have to deal with costs internally, so in this case it performs much better than costgcc, who on the contrary has a higher complexity.
- When the cost upper bound is equal to the minimum, Dom B is best for most instances, while for one instance Val B wins, and for two instances, all costgcc configurations are equivalent.
- Val B is always better than Val, but Dom B is not always better than Dom.
- Val is the worst, except for one instance when it manages to beat Dom and Dom B, because the cost upper bound becomes unrestricted enough to the point that arc consistency cannot prune enough values to outweigh its computational overhead.

The ultimate factor that determines whether costgcc is worth to use over a decomposition or not, is the tightness of the cost sum upper bound. It is easy to see that the closer it is to the minimum cost, the better performance is, and as we loosen it, the overhead of costgcc overtakes and is no longer a good investment. In Table 7, for each instance we show the cost upper bound percentage compared to the minimum one, after which costgcc is no longer beneficial.

Table 7: Cost upper bound relation to min cost, after which costgcc starts to perform worse than decomposition

big	600%
big-sparse	350%
big-var-low-val	550%
big-var-low-val-sparse	155%
low-var-big-val	400%
low-var-big-val-sparse	320%
average	304%

Another interesting point to study is how well we can detect infeasible instances, that is, instances which have zero solutions. For all the datasets presented above, costgcc is remarkable in this regard, as if we reduce the cost upper bound to be lower than the minimum cost, then it reports infeasibility instantly, no matter the value of the bound. The same cannot be said for Multi, which seems to struggle. For example, for instance *lowVarBigVal* which has a minimum cost sum of 100, if we reduce it to anything from 0 to 5 then failure is reported instantly, if we increase it to 5 then it takes 5 seconds, while if we turn it to 7 or more, it does not terminate within the time limit. Or for instance *bigSparse* which has a minimum cost sum of 100, reducing the bound to 20 reports the result instantly, increasing it to 30 takes 7 seconds, while making it 40 or more does not finish within the time cutoff. A similar behaviour is noted with all the rest datasets as well.

We will now tackle a variation of costgcc where it turns to an optimization problem, in which we don't care to just find solutions of cost lower than a given bound, but instead we want to find the solution of minimal cost. We use a Branch and Bound method, whose cost to optimize is the cost upper bound variable in costgcc. Each time a solution is found, we further restrict the cost upper bound, to find solutions of better quality. Referencing the problem case distinction for optimization problems mentioned in Chapter 3.6, this is a problem case of type A.

We run the program and see the results in Table 8. We report the time spent to find the optimal solution, or the distance between the best solution found and the optimal one, in case that we did not manage to find it within a time limit of 5 minutes. On first glance, it is obvious that our branching heuristic is a very strong choice for this type of problems, since both Val B and Dom B find the optimal solution in just a few milliseconds. This is expected, as the heuristic leads us straight to an optimal solution when used on costgcc without any other constraints, as studied in Chapter 3.5.1. Dom is efficient too, taking less than one second for most instances. The execution times for Val are also reasonable. We can clearly see that Multi is the worst configuration, as it never finds an optimal solution.

Table 8: Distance to optimal solution with 5min cutoff for different instances, or time spent in case the optimal solution was found

	Val	$ValB$	Dom	$DomB$	$Multi$
big	405	$72ms$	$6s$	$70ms$	509
big-sparse	$12s$	$44ms$	$0.8s$	$15ms$	505
big-var-low-val	$11s$	$26ms$	$0.9s$	$26ms$	457
big-var-low-val-sparse	$0.3s$	$4ms$	$81ms$	$10ms$	515
low-var-big-val	$7.9s$	$15ms$	$0.2s$	$11ms$	92
low-var-big-val-sparse	$0.1s$	$7ms$	$32ms$	$9ms$	61

5.1.2 Traveling Salesman Problem

The Travelling Saleman Problem (TSP) is a very well known NP-Hard problem, in which the goal is to visit all the cities of an area exactly once each, to return back to the starting city, and to minimize the travelling cost. In graph terms, each node is represented by a city, an edge between two cities is a road that connects them, and there is a cost which is usually defined as the distance between them. To solve the problem, we need to find a Hamiltonian Circle of minimal cost.

There is already a packaged TSP example model in Gecode, which generates Hamiltonian circles using the constraint `circuit` and finds the one of minimum cost using Branch and Bound. In this model, each variable represents a city, and its domain represents the links to the other cities. Assigning variable X_n with value m means that we will include the edge (n, m) in our travel plan. We will call this model *Multi*, and compare it with a model which again uses `circuit` to find Hamiltonian paths, but is also enhanced with a costgcc. The lower and upper bounds of it are equal to one, accounting for visiting each city exactly once. The cost upper bound is the cost variable used for Branch and Bound. As better solutions are found, the internal costgcc cost bound is tightened, which results in stronger propagation. This is also an optimization problem case of type A.

In Table 9, we compare our enhancement with the built-in Gecode approach, across 4 different instances. The instances are taken from TSPLIB [36], which is a library collection of various TSP datasets, a lot of which are derived from real world data. The first two instances are also included in the example packaged in Gecode. Again, we show the time spent to find the optimal solution, or the distance of the best solution found to the optimal one, in case it was not able to be found with a 5 minutes cutoff.

For *br17*, Multi is actually faster than costgcc. For *ftv33* and *bays29*, Dom B finds the optimal solution fast, while Multi reaches a timeout. Across all instances, Dom B is the best configuration for costgcc, and Val B is much better than Val, again showcasing the advantage of our custom branching heuristic.

We also attempted to enhance the Multi model with a gcc to see if there could be any extra propagation that could help, but there was no improvement. In fact it was much

Table 9: Distance to optimal solution with 5min cutoff for different city instances, or time spent in case the optimal solution was found

	Val	$ValB$	Dom	$DomB$	$Multi$
br17	1	$3m26s$	$10s$	$5s$	$10ms$
ftv33	254	61	$2.1s$	$0.6s$	269
bays29	892	392	$1m35s$	$15s$	87
berlin52	11879	2053	3030	989	2287

slower, suggesting that the benefit of costgcc comes from pruning around the cost, and not because of the extra cardinality constraints.

Finally we tested against the Concorde TSP Solver [37]. It is a dedicated software for solving TSP instances efficiently, using various techniques. Concorde is able to optimally solve all the instances in just a few milliseconds. Of course, our approach is not competitive with state-of-the-art algorithms specifically designed for TSP, but it improves the default way to solve it within Gecode.

5.1.3 Warehouse Location Problem

We study the Warehouse Location problem. A company needs to open warehouses in order to supply goods to stores. There is an upper limit to the amount of goods each warehouse can provide (warehouse capacity), and each store can be supplied by only one warehouse. Supplying from a specific warehouse to a specific store has a cost, which usually depends on their distance. There is also a fixed cost tied to opening a warehouse. The goal is to minimize the sum of the costs induced by opening warehouses and the distance costs.

Gecode is packaged with a built-in example of this problem, which is modeled as follows: there is a variable representing each store, and a variable's domain represents the warehouses it can be supplied from. Each variable will be assigned to exactly one value, so with this approach we already comply with each store being supplied from exactly one warehouse. To find out which warehouses are open and to calculate all the necessary costs, `element` constraints are utilized. Finally, a *gcc* constraint is used to express the supply limitation of each warehouse. A Branch and Bound search minimizes the total costs. We will call this model *Multi*.

We attempt a slightly different model in which we use a similar setup, but with costgcc instead of gcc, to see if we can have any benefit from cost pruning reasoning. Here we have an optimization problem of case type B. The total cost to minimize with Branch and Bound is the sum of the open warehouses costs and the costs from assigning warehouses to stores, so the cost upper bound within costgcc is not identical to the Branch and Bound global one, thus we have limited domain reasoning.

Since the hardcoded input data within the bundled Gecode example is trivial to solve, we take datasets from a Discrete Location Problems benchmark library [38]. We look into the Capacitated Facility Location Problem, which is the same as our case problem, but with an extra factor. This version of the problem specifies a *demand* value connected to each store-warehouse pair, which represents the amount of product that will be transferred if said pair is chosen. For each warehouse, the sum of the amount of product concerning it must be less than the warehouse capacity.

First, we ignore the demand values and assume they are all equal to 1, to have the same version of the problem as the one packaged with Gecode, and then we adapt our models to also test the second version. In Table 10 we compare costgcc with Multi, using the first dataset in Class 1 found in the benchmarks section of the Capacitated Facility Location Problem benchmarks library [38].

Table 10: Best solution found with 5min cutoff for the Warehouse Location Problem

	Val	$ValB$	Dom	$DomB$	$Multi$
without demands	7247	4709	7762	4616	6017
with demands	9211	5056	9211	5056	6021

Since the dataset is large, we cannot find the optimal solution with any configuration. However, costgcc manages to get much closer to it than Multi. By trying different instances from the library, we get similar results. Since this is an optimization problem of case type B, we cannot expect to have remarkable results, as the cost upper bound within costgcc is not necessarily tightened enough on each iteration. In addition, the version with the demands is harder than the other one, because we cannot use the value counting part of gcc/costgcc anymore to express it with upper bounds. Because when the demands are all 1, we can just count occurrences of warehouse values, but when not we need to use an external sum constraint.

5.2 Symmetric Global Cardinality

For symgcc, we choose the implementation described in Chapter 4.4.2. We will first run experiments using randomly generated data, and then test our program on **Sports Tournament Scheduling**, which is a real world problem.

5.2.1 Randomly generated instances

We generate data using a separate program, according to the following parameters. All of them are followed by space and then a number, except for `-f` , which is followed by string that specifies the name of the output file.

- `-n`: number of variables
- `-m`: number of values

- `-p`: percentage representing the density of variable-value graph
- `-umin`: minimum value upper bounds percentage
- `-umax`: maximum value upper bounds percentage
- `-lmin`: minimum value lower bounds percentage
- `-lmax`: maximum value lower bounds percentage
- `-uvarmin`: minimum variable cardinality upper bounds percentage
- `-uvarmax`: maximum variable cardinality upper bounds percentage
- `-lvarmin`: minimum variable cardinality lower bounds percentage
- `-lvarmax`: maximum variable cardinality lower bounds percentage
- `-s`: seed for random generator

The value lower and upper bound percentages are in relation to the total occurrences number of a specific value in the domains of the variables, while the variable cardinality lower and upper bound percentages are in relation to the maximum cardinality possible of each variable, based on its domain. If the seed is not specified, then it will be automatically generated and printed, so that identical data can be replicated in the future.

We create the following files with the respective commands, that we will use for experimentation in this chapter:

- big: `./gendata -f big -n 100 -m 100 -p 100 -lmin 0 -lmax 8 -umin 20 -umax 30 -s 3118417791`
- big-sparse: `./gendata -f bigSparse -n 100 -m 100 -p 25 -lmin 0 -lmax 8 -umin 20 -umax 30 -s 3118417791`
- big-var-low-val: `./gendata -f bigVarLowVal -n 100 -m 25 -p 100 -umin 20 -umax 30 -lmin 0 -lmax 8 -s 294367347`
- big-var-low-val-sparse: `./gendata -f bigVarLowValSparse -n 100 -m 25 -p 10 -umin 70 -umax 75 -lmin 50 -lmax 70 -s 294367347`
- low-var-big-val: `./gendata -f lowVarBigVar -n 25 -m 100 -p 100 -umin 20 -umax 30 -lmin 0 -lmax 8 -s 294367347`
- low-var-big-val-sparse: `./gendata -f lowVarBigValSparse -n 25 -m 100 -p 10 -umin 70 -umax 75 -lmin 50 -lmax 70 -s 294367347`

In the above commands we do not include the parameters that control the set cardinality bounds, because they are not constant as we will experiment will various different combinations for them below.

We will compare symgcc with the model using default Gecode constraints described in Chapter 4.1. Between the choice of Gecode's `linear` and `count` constraints to express the sum, we use the former since we have found that it outperforms the latter.

We now present computational results on each of the above datasets. For each instance, we compare our program to Multi for different combinations of the `-uvarmin -uvarmax -lvarmin -lvarmax` parameters. That is, we want to observe how performance changes depending on the cardinality bounds of the variables, as this is the part in which symgcc differs from the original gcc. We keep `-uvarmin` and `-uvarmax` equal to each other as we modify them (and respectively `-lvarmin` and `-lvarmax`). We always run the programs with arc consistency, as we have found that it is the best propagation level for both symgcc and Multi.

In Figures 9, 10 and 11, for each cardinality bounds percentage combination, there is a point that indicates which program managed to find the most solutions within a time limit of 1 minute, or which reported failure faster in the case of absence of solutions. A blue circle indicates that Dom found the most solutions while Multi found none, a red triangle that Multi found the most of them, a green circle that there were no solutions and Dom reported failure instantly while Multi did not manage to do so within the time cutoff, and a black circle that there were no solutions and both programs reported failure right away. Note that by definition the lower cardinality bounds are lower or equal to the upper cardinality ones, and that is why some data points are ”missing” from the plots.

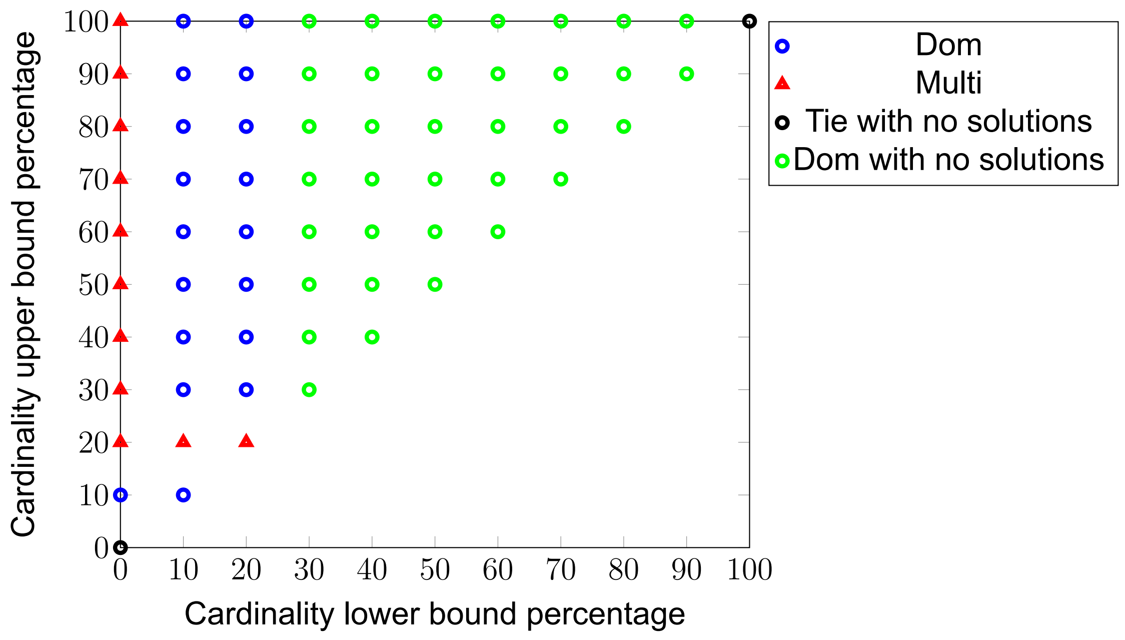

Figure 9: Comparison of symgcc versus Multi for instances *big*, *big-sparse* and *low-var-big-val* for different cardinality bound percentages

In Figure 9, we observe that symgcc outperforms Multi as we tighten the lower cardinality. Even when having the lower cardinality bound as low as 10%, it is enough for Dom to be faster even when the upper bounds are fully loose (aside from 20% and 30% where Multi wins). In the case that we leave the lower cardinality fully unbounded (0%), Multi performs better regardless of the upper cardinality percentage value. We omit the plot for instance *big-var-low-val*, as we have the exact same behaviour as in Figure 9, with only one data point of difference. At coordinates $(10, 30)$, this time Multi wins.

In Figure 10 where the input data results in a sparse graph with many variables but not a lot of values, we see that tightening the lower cardinality seems to be irrelevant for the most part, as it is the upper cardinality that dictates the victor. If we loosen the upper cardinality enough, Multi rises on top, except for when the lower cardinality is tightened more than 50% and the upper is fully loose, in which case the hard lower cardinality bound takes over and lets symgcc win.

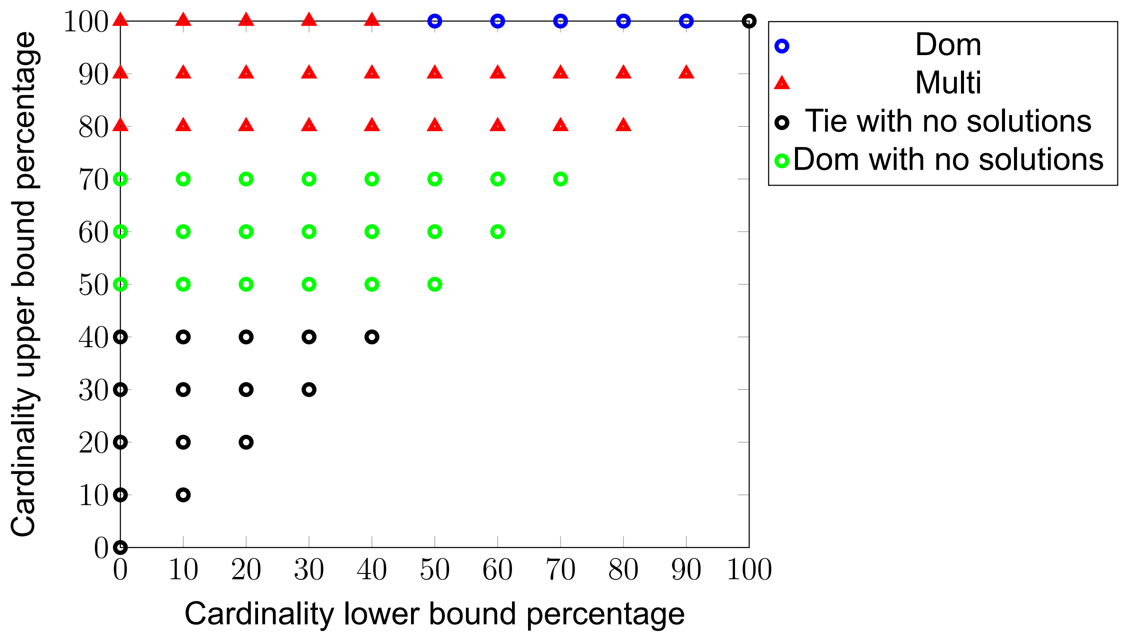

Figure 10: Comparison of symgcc versus Multi for instance *big-var-low-val-sparse*, for different cardinality bound percentages

Finally, in Figure 11 which contains results for a sparse graph instance with many values but not a lot of variables, we observe the following: for lower cardinality bound less than 50%, symgcc wins until the upper cardinality is unbounded enough to be larger than 50%. For lower cardinality bound tighter than 50%, symgcc generally wins regardless of how loose the upper cardinality bound is, except for 4 cases.

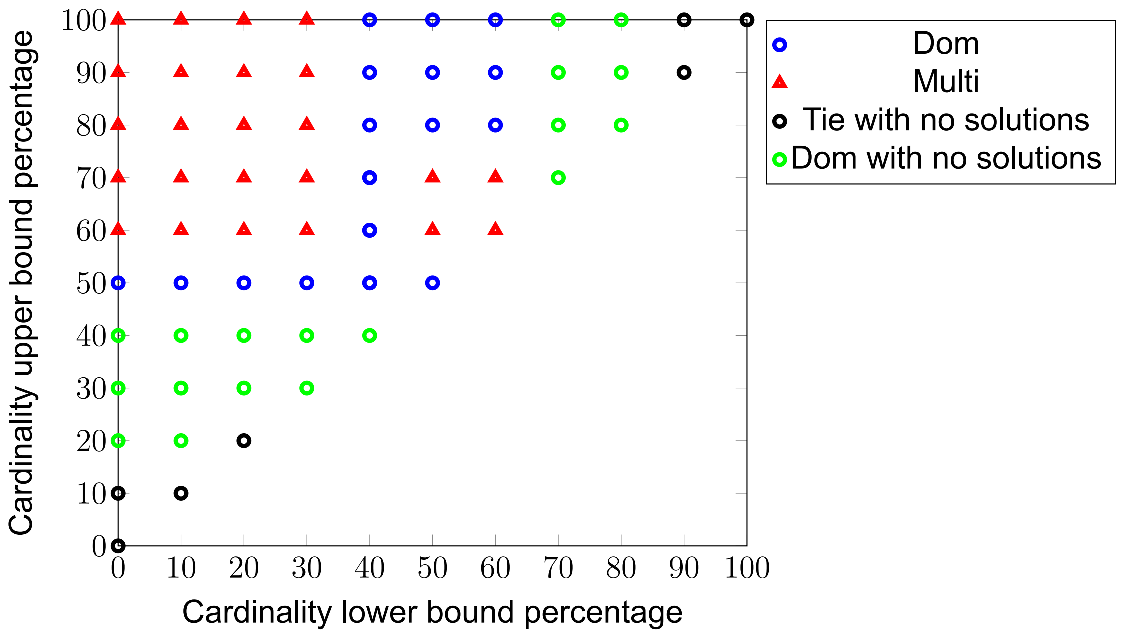

Figure 11: Comparison of symgcc versus Multi for instance *low-var-big-val-sparse*, for different cardinality bound percentages

In general, we can conclude that symgcc can outperform Gecode's default solution as we tighten the cardinality bounds. After all, it's the set cardinality restriction that differentiates symgcc from the original gcc.

5.2.2 Sports Tournament Scheduling

The problem concerns to schedule a tournament involving n teams over $n-1$ weeks, with each week consisting of $\frac{n}{2}$ periods. A period signifies a time slot at which a match between two teams can take place. The goal is to create a schedule such that all teams play with each other once, no team plays twice during each week, and no team plays more than 2 times during each period across the weeks.

To solve this using our constraint, we define a Set variables array of rows equal to the number of periods and columns equal to the number of weeks. Each set variable represents a game between two teams, and thus we set the lower and upper cardinality bounds of all of them to be equal to two.

To state that all teams should play with each other across the tournament season, all we need to do is prohibit a pair of teams from playing with each other more than once. In other words, it suffices to constrain each variable to be different from all the others. For the restriction that no team should play more than once per week, we can define a symgcc for each week, with lower and upper bounds for each value equal to 1. Finally, to ensure that a team can play at most twice during a certain period, we define a symgcc for each period, with the numbers zero and two as lower and upper bounds for each value. For all the symgcc constraints, the cardinality bounds are equal to two, since we are dealing with pairs of teams on each time slot.

Since Gecode does not offer global cardinality constraint defined on sets by default, to achieve the same behaviour without our constraint, we need to double our variables. We now define two variables per time slot, thus resulting in a matrix of rows equal to the number of periods but columns equal to twice the number of weeks. We need to do this to represent pairs of teams.

The restriction about each period remains straightforward, as all we need to do is define a gcc for each, in the same manner as we did earlier. For the restriction per week, we need to make sure that we include all pairs of teams on each gcc for each week. Thus, for each weekly gcc we have double the variables involved. Finally, the obligation that each team should play with each other requires some extra work. Like before, we can express this by stating that no teams should play each other more than once, thus all team pairs should be different.

Assuming w weeks, p periods and t teams, let X be a single dimension version of the matrix that we described that we need for this model, consisting of $w \times p \times 2$ integer variables. If i and j index the beginning of two different time slots within X, we can define the following constraint for each possible pair of i and j representing different time slots:

$$\begin{gathered}\neg(X_i = X_j \wedge X_{i+1} = X_{j+1}) \\ \wedge \\ \neg(X_i = X_{j+1} \wedge X_{i+1} = X_j)\end{gathered}$$

The above states that for a specific pair of time slots, there should not be the same pair of teams playing. We will call this model **Multi**, and we will assume that we use arc consistency on it during experiments, as we have empirically found that it provides the best results for it.

In Table 11, we compare the time required to generate a tournament schedule, for different number of teams and with different configurations. A dash indicates that no solution was found within a time limit of 10 minutes.

Table 11: Time spent to find a solution to the Sports Tournament Scheduling Problem for different number of teams

	Val	Dom	$Multi$
6	$71ms$	$3ms$	$12ms$
8	$2m53s$	$3.9s$	$-$
10	$-$	$6m22s$	$-$

We can clearly see that symgcc outperforms the Gecode alternative. Arc consistency is the best configuration, but even when not applying it, it is still much faster than Multi, as even with 8 teams Multi cannot find a solution within a reasonable time limit. We can conclude that this happens because by using set variables we are dealing with a smaller total number of variables, and propagation can also have a more global impact compared to decomposing to multiple integer variables. It is a problem case in which having the ability to model using a set helps significantly.

6. CONCLUSIONS AND FUTURE WORK

Global cardinality constraints arise very commonly in real world applications. We have studied the constraints Global Cardinality With Costs and Symmetric Global Cardinality and implemented them within the open source constraint solver Gecode, which offered no native support for them prior. We have described various different implementation choices that we have went through, as an attempt to optimize the performance of the constraints. For the Global Cardinality With Costs, we have additionally proposed a powerful branching heuristic that can reduce computational time in most cases. We have discovered through experiments that the more we restrict the cost upper bound for this constraint, the better it will perform compared to using a decomposition. Moreover, we have explained how to expand its use case to the scope of optimization problems, outperforming the Gecode built-in example for the Traveling Salesman Problem. For the Symmetric Cardinality Constraint, it is the tightness of the cardinality bounds that dictates performance. In both cases, we have shown that under certain conditions, using a global constraint that examines the problem fully can provide better results than decomposing to multiple simpler constraints that will potentially miss some domain pruning.

As for future work, there exist many other Global Cardinality related constraints that could be implemented, for instance the Symmetric Cardinality Constraint With Costs, for which we could combine algorithms and ideas from both the programs we have implemented. Furthermore, additional research can be done towards their optimization, for example by attempting a Fibonacci priority queue for Dijkstra's algorithm, or by trying Dynamic Partitioning for Symmetric Global Cardinality, and other micro improvements.

ACRONYMS AND ABBREVIATIONS

CSP	Constraint satisfaction problem
GCC	Global cardinality constraint
EGCC	Extended global cardinality constraint
COSTGCC	Global cardinality constraint with costs
SYMGCC	Symmetric global cardinality constraint
NP	Non-deterministic polynomial time
MRV	Minimum Remaining Values
LCV	Least Constraining Values
DFS	Depth-first search
BAB	Branch-and-bound
TSP	Travelling Salesman Problem
GLB	Greatest Lower Bound
LUB	Least Upper Bound

ANNEX

Figure 12: BestBranch class implementing Local Object Handle to share state between costgcc and custom brancher

```
#ifndef H_BEST_BRANCH
#define H_BEST_BRANCH

#include <gecode/kernel.hh>

using namespace Gecode;

// Local Object Handle for an array of integers on the heap
class BestBranch : public LocalHandle {
protected:
  class LIO : public LocalObject {
  public:
   int* data;
   int n;

   LIO(Space& home, int n0)
    : LocalObject(home), data(heap.alloc<int>(n0)), n(n0) {
     home.notice(*this,AP_DISPOSE);
   }

   LIO(Space& home, LIO& l)
     : LocalObject(home,l), data(heap.alloc<int>(l.n)), n(l.n) {
      heap.copy(data, l.data, l.n);
   }

   virtual LocalObject* copy(Space& home) {
     return new (home) LIO(home,*this);
   }

   virtual size_t dispose(Space& home) {
     home.ignore(*this,AP_DISPOSE);
     heap.free<int>(data,n);
     return sizeof(*this);
   }
  };
public:
  BestBranch(Space& home, int n)
   : LocalHandle(new (home) LIO(home,n)) {}

  BestBranch(const BestBranch& bestBranch)
   : LocalHandle(bestBranch) {}

  BestBranch() {}

  BestBranch& operator =(const BestBranch& bestBranch) {
   return static_cast<BestBranch&>(LocalHandle::operator =(bestBranch));
  }

  int operator [](int i) const {
   return static_cast<const LIO*>(object())->data[i];
  }

  int& operator [](int i) {
   return static_cast<LIO*>(object())->data[i];
  }
};

#endif
```

Figure 13: Custom brancher that reads shared information provided from costgcc to make a good value branching choice

```
#include <gecode/int.hh>

using namespace Gecode;

class BestVal : public Brancher {
protected:
  ViewArray<Int::IntView> x;
  mutable int start;
  BestBranch bestBranch;
  class PosVal : public Choice {
  public:
   int pos; int val;

   PosVal(const BestVal& b, int p, int v)
     : Choice(b,2), pos(p), val(v) {}

   virtual void archive(Archive& e) const {
     Choice::archive(e);
     e << pos << val;
   }
  };
public:
  BestVal(Home home, ViewArray<Int::IntView>& x0, BestBranch& bestBranch)
   : Brancher(home), x(x0), start(0), bestBranch(bestBranch) {}

  static void post(Home home, ViewArray<Int::IntView>& x, BestBranch& bestBranch) {
   (void) new (home) BestVal(home, x, bestBranch);
  }

  virtual size_t dispose(Space& home) {
   (void) Brancher::dispose(home);
   return sizeof(*this);
  }

  BestVal(Space& home, BestVal& b)
   : Brancher(home,b), start(b.start) {
   x.update(home,b.x);
     bestBranch.update(home, b.bestBranch);
  }

  virtual Brancher* copy(Space& home) {
   return new (home) BestVal(home,*this);
  }

  virtual bool status(const Space&) const {
   for (int i=start; i<x.size(); i++)
    if (!x[i].assigned()) {
     start = i; return true;
    }
   return false;
  }

  virtual Choice* choice(Space&) {
   int p = start;
   int maxRegret = x[start].regret_max();
   for (int i=start+1; i<x.size(); i++) {
        int regret;
        if (!x[i].assigned() && ((regret = x[i].regret_max()) > maxRegret)) {
          p = i; maxRegret = regret;
        }
     }
   return new PosVal(*this,p, bestBranch[p]);
  }

  virtual Choice* choice(const Space&, Archive& e) {
   int pos, val;
   e >> pos >> val;
   return new PosVal(*this, pos, val);
  }

  virtual ExecStatus commit(Space& home,
                            const Choice& c,
                            unsigned int a) {
   const PosVal& pv = static_cast<const PosVal&>(c);
   int pos=pv.pos, val=pv.val;
   if (a == 0)
    return me_failed(x[pos].eq(home,val)) ? ES_FAILED : ES_OK;
   else
    return me_failed(x[pos].nq(home,val)) ? ES_FAILED : ES_OK;
  }

  virtual void print(const Space&, const Choice& c, unsigned int a,
     std::ostream& o) const {
   const PosVal& pv = static_cast<const PosVal&>(c);
   int pos=pv.pos, val=pv.val;
   if (a == 0)
    o << "x[" << pos << "] = " << val;
   else
    o << "x[" << pos << "] != " << val;
  }
};

void branchBestVal(Home home, const IntVarArgs& x, BestBranch&
    bestBranch) {
  if (home.failed()) return;
  ViewArray<Int::IntView> y(home,x);
  BestVal::post(home, y, bestBranch);
}
```

REFERENCES

[1] *Gecode* <`https://www.gecode.org/index.html`> [accessed 5 April 2022]

[2] J-C. Régin, *Cost-Based Arc Consistency for Global Cardinality Constraints*, Constraints 7, 2002, pp. 387-405

[3] J-C. Régin, *Global Constraints: a Survey*, Hybrid Optimization, pp. 63-134, 2010

[4] T. Petit, J-C. Régin, C. Bessière *Specific filtering algorithms for overconstrained problems*, Principles and Practice of Constraint Programming - CP 2001

[5] C. Gervet *Programmation par Contraintes sur Domaines Ensemblistes*, Habilitation à diriger des Recherches, Université de Nice-Sophia Antipolis, 2006.

[6] WJ. van Hoeve, J-C. Régin, *Open Constraints in a Closed World*, Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 2006

[7] K. McAloon, C. Tretkoff, G. Wetze, *Sports League Scheduling*, Proceedings of the 3th Ilog International Users Meeting, 1997

[8] B.D. Parrello, W.C. Kabat, L. Wos, *Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem*, J Autom Reasoning 2, pp. 1-42, 1986

[9] S. Huczynska, P. McKay, I. Miguel, P. Nightingale, *Modelling Equidistant Frequency Permutation Arrays: An Application of Constraints to Mathematics*, Principles and Practice of Constraint Programming - CP 2009

[10] J-C. Régin, *Generalized arc consistency for global cardinality constraint*, Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, Volume 1, August 4-8, 1996,

[11] L.R. Ford, D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, 2010

[12] Tarjan R. *Depth-first search and linear graphs algorithms*, 2nd Annual Symposium on Switching and Automata Theory, pp. 114-121, 1971

[13] C-G Quimper, A. López-Ortiz, P. van Beek, A. Golynski, *Improved Algorithms for the Global Cardinality Constraint*, Principles and Practice of Constraint Programming - CP 2004

[14] P. Nightingale *The extended global cardinality constraint: An empirical survey*, Artificial Intelligence 175(2), pp. 586-614, 2011

[15] I. Katriel, S. Thiel *Complete Bound Consistency for the Global Cardinality Constraint*, Constraints 10(3), 2005

[16] C-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint*, Constraints 10(2), pp. 115-135, 2005

[17] C-G Quimper, *Efficient Propagators for Global Constraints*, PhD Thesis, University of Waterloo, Canada, 2006

[18] W. Kocjan, P. Kreuger, *Filtering Methods for Symmetric Cardinality Constraint*, Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004

[19] W. Kocjan, P. Kreuger, B. Lisper, *Symmetric Cardinality Constraint With Costs*, MRTC Report, Malardalen University, 2004

[20] R. Cymer, *Applications of Matching Theory in Constraint Programming*, Thesis, Gottfried Wilhelm

Leibniz University of Hanover, 2013

[21] T. Petit, J-C. Régin, *The Ordered Distribute Constraint*, International Journal of Artificial Intelligence Tools 20(04), 2012

[22] N. Beldiceanu, I. Katriel, S. Thiel, *GCC-like Restrictions on the Same Constraint*, Joint ERCIM/-CoLogNet International Workshop on Constraint Solving, volume 3419, 2005

[23] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows* Prentice Hall, 1993

[24] C. Berge, *Graphe et Hypergraphes*. Dunod, Paris, 1970

[25] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976

[26] R.E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, 1983

[27] *ECLiPSe* <`http://eclipseclp.org/index.html`> [accessed 5 April 2022]

[28] Y. Caseau, P-Y. Guillo, E. Levenez, *A deductive and object-oriented approach to a complex scheduling problem*, Proceedings of DOOD'93, 1993

[29] S.V. Cauwelaert, P. Schaus, *Efficient Filtering for the Resource-Cost AllDifferent Constraint*, Constraints Volume 22, pp. 493-511, 2017

[30] *gecode-gcc-extensions* <`https://github.com/iPapatsoris/gecode-gcc-extensions`> [accessed 7 August 2022]

[31] E. W. Dijkstra , *A note on two problems in connexion with graphs*, Numerische Mathematik volume 1, pp. 269-271, 1959

[32] R. Bellman, *On a routing problem*, Quarterly of Applied Mathematics 16, pp. 87-90, 1958

[33] E.F Moore, *The shortest path through a maze*, Bell Telephone System, 1959

[34] J. Yen *An algorithm for finding shortest routes from all source nodes to a given destination in general networks*, Quarterly of Applied Mathematics 27, pp. 526-530, 1970

[35] M. Bannister, D. Eppstein *Randomized Speedup of the Bellman-Ford Algorithm*, Proceedings of the Meeting on Analytic Algorithmics and Combinatoric, pp. 41-47, 2012

[36] *TSPLIB* <`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`> [accessed 10 July 2022]

[37] *Concorde TSP Solver* <`https://www.math.uwaterloo.ca/tsp/concorde.html`> [accessed 10 July 2022]

[38] *Discrete Location Problems Benchmarks* <`http://www.math.nsc.ru/AP/benchmarks/english.html`> [accessed 10 July 2022]