NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES
COMPUTER SCIENCE

MASTER THESIS

Extending upon the Gecode open source toolkit, for developing constraint-based systems and applications

Ioannis A. Papatsoris

Supervisor: **Panagiotis Stamatopoulos,** Assistant Professor

ATHENS

APRIL 2022

MASTER THESIS

Extending upon the Gecode open source toolkit, for developing constraint-based systems and applications

Ioannis A. Papatsoris
R.N.: CS3180006

April 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΛΗΡΟΦΟΡΙΚΗ

PROGRAM OF POSTGRADUATE STUDIES
COMPUTER SCIENCE

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επεκτείνοντας την πλατφόρμα ανοιχτού κώδικα Gecode, για την ανάπτυξη προβλημάτων ικανοποίησης περιορισμών

Ιωάννης Α. Παπατσώρης

Επιβλέπων: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επεκτείνοντας την πλατφόρμα ανοιχτού κώδικα Gecode, για την ανάπτυξη προβλημάτων ικανοποίησης περιορισμών

Ιωάννης Α. Παπατσώρης
Α.Μ.: CS3180006

ΕΠΙΒΛΕΠΩΝ: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής
Κωνσταντίνος Χατζηκοκολάκης, Αναπληρωτής Καθηγητής

Απρίλιος 2022

ABSTRACT

SUBJECT AREA: Constraint Programming

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

ΠΕΡΙΛΗΨΗ

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Θεωρία Γράφων

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

Dedications here

ACKNOWLEDGMENTS

Acknowledgments here

CONTENTS

LIST OF FIGURES

PREFACE

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 conjuction 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. 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 with costs constraint, describing our implementation on Gecode and providing experimental results, comparing it with a model using built-in Gecode constraints.
3. In Chapter 4 we discuss the symmetric global cardinality constraint, our implementation on Gecode, and compare it with a model that combines Gecode's built-in constraints.
4. In Chapter 5 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 $1 \leq j, k \leq N$ with $j \neq k$
4. Constrain $|i - j| \neq |X_i - X_j|$ for $1 \leq j, k \leq N$ with $j \neq k$

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.

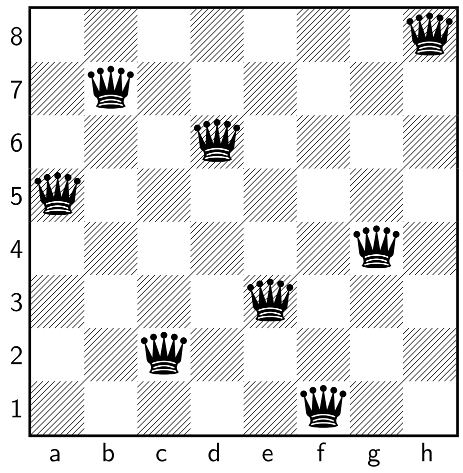

Figure 1: A solution to the 8-Queens problem

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 pairwise 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 $O(|X(C)|^2 D(X) + |X(C)|^{2.66})$

- **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 conjuctions 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 path** 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 occurences 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

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

3.1 Modeling With Gecode

Costgcc can be decomposed into the conjuction 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 i with value 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 prunning 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.
- **B**: Controls whether to use the custom branching technique described in **??** 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`.

Using the argument B for the custom branching described in Chapter **??** is **highly recommended**, as we will show in Chapter 3.7 that it dramatically improves performance. In fact, it is always beneficial to use, unless a better custom branching solution exists that is specialized for a particular problem that uses more constraints other than costgcc. Even in this case, it should be at least benchmarked to see which approach is better.

`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 a complete example of usage and the necessary code, see Figure **??**.

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.

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 do is 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 occurences of said value u in the solution, and thus it will respect the corresponding lower and upper bounds specified by gcc for this value.

For the case of costgcc, all we need to do is find a minimum cost flow with cost less than or equal to the cost upper bound. 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 contraint 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 theoritical 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 exlcude 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 containg 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 care about 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

While the publication of Régin is complete in regards to the algorithm presentation from a theoritical 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 uppper 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. For this check, 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 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.

Instead of deleting the edges every time, we keep 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 transfering 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.

** TODO: try to fix shared handle 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. Unfortunately this comes at cost, as there is a noticeable performance drop by switching from raw pointers to smart ones.

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 [30], 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 [31] 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** [32], 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, and not in the arc consistency part. For the arc consistency, we calculate the reduced costs on the spot and use Dijkstra's algorithm, which proves to be faster 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 if find 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.

**TODO: try it 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 [33, 34] to the Shortest Path Faster Algorithm exist that can reduce the worst case number of itereations 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 structures:

- `list`: a vector holding its adjacent nodes, along with any additional data like edge bounds and flow value.
- `valToPos`: a hashtable 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.

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

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 (by strict definition of the residual graph in chapter 2.2.2, they 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 scheudle 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 elegancy of that structure are based on the fact that it is restricited 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 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, since we would have no way to efficiently save and restore the costs. But now that we 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.

3.5 Custom Branching Heuristic

In the practical improvements section of the original paper by Régin, the use of the Max Regret branching heuristic is suggested, because due to the nature of the prunning algorithm, we have the ability to precisly compute the regret value, instead of approximate it, like it is usually the case with other problems. However, this heuristic did not prove to be useful in practise. Instead, we propose a different one, which dramatically reduces the number of failed search nodes and improves running time by a large factor.

The costgcc find 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 guruanteed 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 conjuction 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 contraint. 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 impove it by looking for one with smaller cost. This method will work also in conjuction 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 varible that we wish to minimize depends on costgcc and external constraints.

In both cases, muptiple constraints can co-exist with costgcc, but the important part is how the cost variable is contrained. 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 3.7. 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.

3.7 Experimental Evaluation

mention that we use smart-bt in section tade which is tade with SPFA which is the best config. we compare val valB dom domB.

3.7.1 Randomly generated instances

Internally, Gecode's gcc bound consistency is based on [16], and domain consistency is based on [10].

3.7.2 Traveling Salesman Problem

3.7.3 Warehouse Location Problem

4. SYMMETRIC GLOBAL CARDINALITY

4.1 Modeling With Gecode

4.2 Constraint Usage

4.3 Algorithm

4.4 Implementation Details

4.4.1 Graph Structure

4.4.2 Incrementality

4.5 Experimental Evaluation

5. CONCLUSIONS AND FUTURE WORK

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
DFS	Depth-first search
LDS	Limited descrepancy search
BAB	Branch-and-bound
AFC	Accumulated failure count
CHB	Conflict-history based branching

ANNEX

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

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

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

[33] 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

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