A General Purpose Local Search Solver

January 27, 2016

Contents

1 Introduction

The field of optimization can be split into several subfields depending on the nature of the decision variables (continuous, discrete) and the structure of the problem (linear, non linear, combinatorial, convex, non convex). The main focus of this thesis is discrete optimization both linear and non linear. Several solvers are available for discrete optimization. The mixed integer linear programming (MILP) approach has solvers such as *GLPK*, *Gurobi* and *CPLEX*. There are also constraint programming (CP) solvers, such as *Gecode*. All these solvers solve the problem exactly, but for some problems thit is not always possible due to the computational cost. Another approach is to use local search and find a good solution fast by making a trade off between speed and solution quality.

There exists a vast literature about how to make good local search solvers for specific problems. However, only a few attempts have been made to use local search for general purpose solvers like mathematical programming and constraint programming. *Comet* was a successful CP based solver and that allowed to use local search to find a good solution fast but the project is now abandoned.
OscaR is another CP based solver that uses local search to find a good solution. **(OscaR) (Overall made)** In this project, a general heuristic solver based on local search has been developed. It uses Gecode to find an initial solution and uses local search trying to improve the solution. It can solve problems formulated as binary programming problems but can be extended to solve a wider range of problems. Ideas for structuring the framework are drawn from Comet, OscaR, and Gecode.

(More specific) Beside the basic components of local search several elements have been studied to see their effect on the solution. One of these elements is preprocessing, with the help from Gecode, that can reduce the size of the search space before the local search is started. Other elements that have be studied are invariants and a directed acyclic graphs to represent efficiently the dependencies between the variables and invariants. The choice of neighborhoods and how to efficiently explore these neighborhoods have been considered. The quality of a solution is evaluated by a vector in lexicographic order instead of a single value. The lexicographic order is from a priority given to the constraints that will affect the local search. Finally, on top of the local search different combinations of neighborhoods, procedures, and metaheuristics has been tested.

The framework has a solid base from which it can be extended to solve a wide range of problems by implementing different constraints and new neighborhoods.

The performance of the solver has been tested with the instances from the MIPLIB 2010 and compared to Gurobi.

2 Discrete Optimization

2.1 Variables

Discrete optimization models contain a set of n variables $X = \{x_1, x_2, \ldots, x_n\}$ and let $I = \{1, 2, \ldots, i, \ldots, n\}$ be the set of indices of X. Each variable $x_i \in X$ has a *domain* $D(x_i) \in D$ where D is the Cartesian product of n domains $D = D_1 \times D_2 \times \cdots \times D_n$ such that $x_i \in D_i$. The variables $x_i \in X$ of the models that will be discussed in this thesis all have their domain restricted to a finite discrete domain $D_i \subseteq \mathbb{Z} \ : \forall i$. The value of a variable x is denoted $V(x)$. The number of constraints that apply to a variable x_i is the *degree* of the variable $deg(x_i)$. We say a variable is independent if the value is allowed to change within its domain in contrast to a dependent variable that only changes dependent on other variables change.

2.2 Constraints

The values of variables will be restricted by a set of m constraints $C = \{c_1, c_2, \ldots, c_j, \ldots, c_m\}$ and let $J = \{1, 2, \ldots, j, \ldots, m\}$. The set of variables to which the constraint c_j applies is called its *scope* and is denoted $X(c_j) = \{x_{1j}, x_{2j}, \ldots, x_{\alpha_j j}\}$. The variable x_{ij} is the i'th variable in constriant c_j and corresponds to a variable $x_k \in X$. The size of a scope $|X(c_j)|$ is called the *arity* α_j. If all variables of a constraint c_j has a finite domain then the constraint can be written in extensional form. *Extensional form* is a subset of the Cartesian product of the domains of the variables in its scope $X(c_j)$, i.e, $c_j \subseteq D(x_{1,j}) \times D(x_{2,j}) \times \cdots \times D(x_{\alpha_j,j}),\ x_{ij} \in X(c_j)$, that is the set of tuples that satisfy the constraint c_j.
We call a constraint c_j a *functional constraint* if given an assignment of values to all variables except x_i in c_j, then at most one value of x_i satisfy c_j for all $x_i \in X(c_j)$. In other words the value of a variable in a functional constraint can be determined from the values of the other variables in the functional constraint.

2.3 Problem Formulation

A *Constraint Satisfaction Problem* (CSP) is defined as a triple $\mathbb{P} = \langle X, D, C \rangle$.
A *candidate solution* to a CSP $\mathbb{P}$ is a vector of n elements
$\tau = (V(x_1), V(x_2), \ldots, V(x_n))$ from the set of all candidate solutions S called the *search space*. Given a sequence $X' \subseteq X$ of variables $\tau[X']$ is called a restriction on τ, ordered according to X. The constraint c_j is satisfied by τ if the restriction $\tau[X(c_j)]$ matches a tuple of the constraint c_j in extensional form. If each constraint $c_j \in C$ is satisfied then the solution τ is a *feasible solution* to the CSP $\mathbb{P}$.

2.4 Solutions

For a CSP the questions of interest could be to report all feasible solutions $sol(P) \subseteq S$, any feasible solution $\tau \in sol(\mathbb{P})$ or if there exists a feasible solution τ or not.
The CSP $\mathbb{P}$ can be expanded to a *Constraint Optimization Problem* (COP) $\mathbb{P}'$ with an evaluation function $f(\tau) \to \mathbb{R}$ that evaluates the quality of the solution τ, $\mathbb{P}' = \langle X, D, C, f) \rangle$. In the COP the task is then to find a feasible solution $\hat{\tau}$ in S such that $f(\hat{\tau}) \leq f(\tau)$ for all feasible solutions τ of $\mathbb{P}$ if it is a minimization problem.

3 General Purpose Solution Methods

3.1 Binary- and Integer Linear Programming

Binary- and integer linear programming can be used to model a wide range of problems by posting linear constraints and using and a linear objective function. A integer linear program (ILP) can be writing on the form:

$$\text{Minimize} \quad z = \mathbf{c}^T\mathbf{x} \tag{1}$$
$$\text{subject to} \quad A\mathbf{x} \leq \mathbf{b} \tag{2}$$
$$\mathbf{x} \in \mathbb{Z}^n \tag{3}$$

Here A is a $n \times m$ matrix of coefficients, $\mathbf{b} \in \mathbb{R}^m$, z is the value of the objective function and $\mathbf{c} \in \mathbb{R}^n$. The first line is the objective function and can easily be transformed to a maximization problem by multiplying by -1. The relation in line 2 can be a mix of $\{\leq, =, \geq\}$ but greater than or equal can be transformed to less than or equal by multiplying both side of the constraint by -1.
A candidate solution is an assignment of values to all variables $\mathbf{x}$ and a solution is said to be feasible if all constraints are satisfied. The set of feasible solutions consist of integer point in a n dimensional space and the point that minimize the objective function is said to be the optimal solution. There can be multiple optimal solutions for a given model.
Solving a general integer program or binary program is a NP-hard problem [4, p.30] and several techniques are developed for solving them. The techniques can be i.e. branch and bound, cutting plane and branch and cut [4, p.31]. An integer linear program can be relaxed by relaxing the integer constraint, line 3 changed to $\mathbf{x} \in \mathbb{R}^n$, on the variables creating a linear programming problem [4, p. 30]. The linear programming problem is easier to solver than the integer programming problem and can be used to find bounds on the integer programming problem. The linear formulation can be transformed to *standard form*, all constraints are equality constraints, by introducing auxiliary variables and then solved using the simplex method.
There exist several solver for (integer) linear programming problems such as Cplex, Gurobi, GLPK and Scip.

3.2 Constraint Programming

Constraint programming involves defining variables and constraints like ILP but often a wide range of constraints can be used. Constraint programming is declarative programming that is the program describe the desired result and not the commands or steps to reach it, just like ILP. Constraint programming uses some interface either a programming language or a framework that have procedures implement to solve the problem according to the

constraints posted. The language or framework may provides global constraint that can be used to formulate the problem. An example of a global constraint is the alldifferent(**x**) constraint that specify the variables **x** must have pairwise distinct values.
Two important aspects of solving a CSP are inference and search. *Inference* is adding constraints to the CSP that does not eliminate any feasible solution but might make it easier to solve the CSP [1, p.301]. Local constraint propagation is an example of inference when dealing with variables with finite domain and can be used to eliminate large subspaces of the search space S. Propagation can be restricting domains of variables, called filtering, or combinations of values to variables, based on the constraint doing propagation [1, p. 169]. Propagation can be done when a constraint is created by eliminating values from the domain of variables. I.e. by doing propagation on the constraint $x_1 + x_2 \leq 2$ where $x_1, x_2 \in \mathbf{Z}^+$ we can reduce the domain of the variables to $\{0, 1, 2\}$. If we set $x_1 = 1$ we can again do propagation and reduce the domain of x_2 to $\{0, 1\}$.
Search strategies explores possible assignments of variables and an exhaustive search would be a combination of all possible assignments of values to the variables. When combining propagation and search strategies the search space can be examined exhaustively and large subspaces can be pruned by propagation.
COP

3.2.1 Gecode

Gecode (GEneric COnstraint Development Environment) is a constraint programming solver implemented in C++ and offer a wide range of modeling features. Gecode offers more than 70 constraints from the "Global Constraint Catalog" [6] that can be applied to boolean, integer, set and float variables. **(Muligvis Gecode architecture billede)**
A model created for Gecode is created by inheriting the **Space** class. **Space** is a class in Gecode that a user can create the model in. To create variables or post constraints the user need to specify the **Space** they should be created in. When variables are created in a **Space**, views are created and associated with the variables. Views are not used in modeling but are used to know when propagation should be made on a constraint.
When posting constraints in a **Space**, Gecode creates propagators and these propagators can subscribe to the views of the variables in the constraint. When variables changes domain the corresponding view tell its subscribes that the variables domain has changed. For some constraint the user has the option to choose the propagator based on a consistency level. The cost of different consistency level varies from linear $O(n)$ in the number of variables n to exponential $O(D(x)^n$. [7, p.57].
To solve a problem Gecode needs guidance when searching and that is done

by a branch method. Once a problem has been formulated, the user must define on which variables and how branching is done. Just like variables and constraints are posted in a **Space** the branch order is also posted on the **Space**. The choices in branching for a set of variables are which of those variables to branch on first and what values to branch on. One can post several branch methods and they are treated in the order they are posted. Once all variables have been branched in one branch method it continues with the next branch or restarts. If no branch strategy is chosen for a variable then branching is not done on that variable.
To start the search a Gecode search engine must be chosen and Gecode offers two, a depth first search engine and a branch and bound engine. Search engines have an option class in which several options can be set [7, p.157].
When searching for a solution in a **Space**, the search can be illustrated as a binary tree where the edges are branch choices for a variable and the vertices are the **Space** created because of those choices. If it reaches a point where no feasible solution is possible it stops branching from that vertex and the **Space** is said to be failed. Gecode creates clones of **Space**s while searching for a solution that are used for backtracking. When Gecode reaches a failed **Space**, instead of starting from scratch and recompute all way to down to the previous vertex, it uses the closest clone to backtrack to that **Space**.

3.3 Heuristics and Local Search

In contrast to integer programming and constraint programming, local search does not do an exhaustive search for a solution. Local search is based on making small changes to the current solution and see if it can improve and sacrifices the optimality guaranty for performance.
To create a model for local search a set of variables and a set constraints for the variables needs to be defined. The constraints are either implicit constraints or soft constraints. The variables and implicit constraints define the candidate solution and hence the search space S.

3.3.1 Implicit Constraints

Implicit constrains are constraints that, once satisfied, always stay satisfied during local search. Each neighborhood operation is made in a way that implicit constraints are kept satisfied.

3.3.2 Soft Constraints

Soft constraints are constraints that the local search should try to satisfy when making neigborhood operations. If a soft constraint is not satisfied we say it is violated and it contributes to the evaluation function. There are different ways of measuring the violation in a constraint. One way is

a binary value, zero for not violated and one for violated. Another way is to use *violation degree*, a measurement of how violated a constraint is, by a function depending on the constraint. I.e. for the **alldifferent(X)** constraint, that is all variables **X** must have pairwise distinct values, it can be the number of variables with the same value as another variable.

3.3.3 Basic Aspects of Local Search

There are several important aspects of local search and the most basic will be covered here.
The *evaluation function* $f(\tau)$ evaluate the quality of the candidate solution. The *neighborhood function* $N(\tau)$ that defines the candidate solutions $s \in S$ close to a given candidate solution τ. The set of candidate solutions s is said to be the neighborhood of τ and can be reach by using the neighborhood function once, called a *neighborhood operation*. We call it an iteration each time we move from one solution to another and Several solution might be explored before making a neighborhood operation. The cardinality of $N(\tau)$ is called the neighborhood size of τ. The neighborhood function is symmetric if, $\tau' \subseteq N(\tau)$ if and only if $\tau \subseteq N(\tau')$ for all pairs of τ and τ'. In a problem with n binary variables the neighborhood function could be to change the value of a single variable, called a flip. This can be done on all variable hence the neighborhood size of a candidate solution would be n.
It might be expensive to use the evaluation function to evaluate each solutions, instead a *delta evaluation function* $\delta(\tau)$ can be used. The evaluation function recomputes the quality of a solution even if only a few variables changes value. The delta evaluation function only computes how much the value of the evaluation function will change by going from one solution τ to another solution τ', hence $\delta(\tau) = f(\tau') - f(\tau)$.
The search space combined with the neighborhood function can be illustrated as a graph $G = (V, E)$. The set V is a set of vertices each representing a candidate solutions of the search space S. The set E is a set of edges connecting a vertex v, representing the solution τ, with the vertices representing the solutions in $N(\tau)$. The graph G is called the *neighborhood graph* and local search does a walk through the neighborhood graph when searching for a solution. [5, p. 3-5]
Local search needs a *termination criteria* that determines when the search is stopped. Sometimes we know what the optimal solution is, i.e. the SAT problem, once we find a feasible solution we can stop. In other cases an optimal solution may not be know, several combinatorial problems are not solved to optimality. The termination function can i.e. be based on a time limit, number of steps made, or when a locally optimal solution is found. A locally optimal solution for a minimization problem is a solution $\hat{\tau}$, such that for each feasible solution $\tau \in N(\hat{\tau})$ $f(\hat{\tau}) \leq f(\tau)$.

3.3.4 Local Search Algorithms

When a local search algorithm makes a neighborhood operation from a current solution to a new solution we say it commit the neighborhood operation. One of the basic local search algorithms is the *iterative improvement* with different pivot rules. Iterative improvement explores the neighborhood of the current solution or a subset of the neighborhood and uses the delta evaluation function to determine which of the neighborhood operations to commit.
Two basic pivoting rules for iterative improvement are *Best improvement* and *first improvement*. Best improvement examine all solutions in the neighborhood of the current solution with delta evaluation function and chooses the solution that gives the best improvement, if any. First improvement examine the solutions in the neighborhood and commits the first neighborhood operation that gives an improvement, if any. iterative improvement is repeated until no improving solution exists.
Several heuristics can be applied to the algorithms for instance by choosing a subset of the neighborhood to examine at random when using best improvement or allowing a number of consecutive sidewalks. A sidewalk is going from a solution τ to a neighbor solution τ' where $\delta(\tau') = 0$. First improvement can be modified to *random improvement* that chooses an improving solution with a probability p or looks for the next improving solution with probability $1 - p$.
Another basic local search algorithm is the *random walk*. Random walk commits a neighborhood operation chosen uniformly random and repeats that for number of iterations. This might lead to worsening solution and/or infeasible solution and is usually combined with other heuristics or local search algorithms.
Before local search can be done an initial assignment to the variables is needed and an initialization function is used for this also called a construction heuristic.

3.3.5 Construction Heuristics

A construction heuristics is used to find an initial candidate solution to a given instance. One of the main things to consider when creating a construction heuristic algorithm is the balance between quality of the solution and time complexity of the algorithm. The extreme case would be to solve the given instance with a construction heuristic but then the main point of local search is lost, that is finding a high quality solution fast.
The first thing to consider when creating a construction heuristic is whether it has to find a feasible solution or it is allowed to find an infeasible solution.

The choice depends on how the local search is designed, whether it can reach infeasible solution or not.
An example of a simple construction heuristic is creating a random candidate solution. For each variable a value between its lower and upper bound is chosen uniformly at random. This is a fast construction heuristic $O(n)$ but cannot give any guarantee of the quality. Examples of other construction heuristics can be greedy heuristics like first fit or best fit.
Construction heuristics can be tailor made to a specific problem type to find a good initial solution depending on the problem. It can be beneficial to introduce randomness in the algorithm and rerun it a couple of times and to get a better initial solution.

3.3.6 Metaheuristics

Metaheuristics defines how the search space should be explored in where as local search algorithms that focus more on the neighborhood of the solution. Iterative improvement is often fast to find a local optimum depending on the size of the neighborhood. Usually only a small fraction of local optima are close to optimality and they can be a poor quality solution [5, p.135]. Metaheuristics are used to get out of local optima to search different parts of the search space.
One of the simple Metaheuristics is *iterative local search* that remembers the best solution found so far and uses two local search algorithms, random walk and iterative improvement. It uses iterative improvement to find a local optima and compare it to the best solution so far. Then uses random walk for some iterations to escape the local optima. These two algorithms are repeated until the termination criteria is reached.
Another metaheuristics is *tabu search* that implements an iterative improvement with a modified best improvement pivoting rule. It chooses the neighborhood operation that will leads to the best solution in the neighborhood but not necessarily an improving solution. In addition to the modification of the pivot rule it implements a tabu list T that keeps track of the last t solution. The solution in the tabu list are not consider when looking for the next solution. It might be very memory expensive to keep track of the last t solutions. An alternative way of implementing the tabu list is to keep track of the last t neighborhood operations and forbid the reverse neighborhood operation. This is more restrictive since the reverse neighborhood operations might not lead to a solution visited within the last t iterations. To compensate for this a *aspiration criteria* can be implemented. Aspiration criteria is a set of rules that can overrule the tabu list [5, p.139-140]. A common aspiration criteria is to ignore the tabulist if the neighborhood operation leads to the best solution found so far.
Several other metaheuristics exist such as variable neighborhood search and very large scale neighborhood search. Variable neighborhood search uses

different neighborhood functions, hence different neighborhoods. Very large scale neighborhood search changes many variable in each neighborhood operation which give a very large neighborhood but might need fewer iteration to find a good quality solution.

3.3.7 Creating a Model for Local Search

When creating a model for local search there is several things to consider, most important things to consider are what the variables should represent and what constraint should be imposed on them. The choice of variables and constraints together with the step function should be such that the delta evaluation can be calculated fast, preferably $O(1)$. The step function should be chosen such that the search space can be explored efficiently. One thing to consider is whether allowing candidate solution that are infeasible or not, once a feasible solution is found.

3.4 Constraint Based Local Search

Constraint based local search (CBLS) is trying to combine the concept of constraint programming and local search. Constraint programming gives a natural way of describing a problem and can reuse the global constraint, propagators, and search strategies. Local search has concepts such as moves, neighborhood, and metaheuristics that have specific implementations for each problem type. Local search offers high quality solution within a relatively short time limit.
The idea of a CBLS framework is to offer global constraint to formulate the problem while local search algorithms are used to solve the model. The user can focus on modeling their problem instead of creating and optimizing algorithms to solve it. This gives the reusability and formulation power of constraint programming while having the performance of local search.
To make local search efficiently on the constraints formulated new data structures are introduced such as Invariants and oneway constraints.

3.4.1 Invariants and Oneway Constraints

Invariants are dependent variables, whose value are functionally defined by other independent variables and/or invariants. Invariants can represent variables or auxiliary variables whose value are of interest.
One-way constraints are constraints that functionally defines the value of an invariant.

$$V(y) = f(X'),\ X' \subseteq X \tag{4}$$

Where $f(X')$ is a oneway constraint with the scope X' that functionally defines the invariant y. The operators for an invariant y are the same as

those for a variable x such as $V(y)$ is the value of the invariant.

3.4.2 Evaluating Solutions

There are different ways of measuring the quality of an solution. For a CSP a feasible solution needs to be found but different feasible solutions are equally good.
In a COP the feasible solutions are compared by an objective function that gives the feasible solution a measurement of quality.
There are different ways of differentiating infeasible solution from other solutions, feasible or infeasible. The evaluation function is a combination of the objective function and the violated constraints.
Each constraint can be given a weight and together with the violation degree it contribute to the evaluation function. I.e. a constraint has weight ten and a violation degree of three it then contribute $3 \cdot 10 = 30$ to the evaluation function. In this way the search priorities to satisfy certain constraints more than other. This method can have the negative effect that it might priorities to have all constraint with violation degree close to zero instead of one constraint with very high violation degree. If many constraint are violated it might take more iterations to get to feasible solution than if only a single constraint is infeasible.
Another way it to use a priority for each constraint as a lexicographic ordering, and try to satisfy constraints in that order. Once a group of constraints with priority p is feasible they are always kept feasible and it tries to satisfy constraint of priority $p - 1$. This will reduce the search space each time a group of constraints is satisfied and might lead to a scenario where the optimal solution cannot be reached.
This framework introduce a third way that uses priorities of constraints for a lexicographic ordering. Instead of evaluation a single value a vector of values from each solution can be compared in lexicographic order. The values representing violations of constraints of priority p is first compared if equal it moves to $p - 1$ and so on. The first vector to have a smaller value is considered the best. This prevent getting trapped in a subset of the search space S.
The vectors are never compared implementation wise but the delta evaluation function returns a delta vector that is used for evaluation instead of a delta value. This will be discussed further in section 8.

3.4.3 Comet

Comet is an object oriented programming language that uses the modeling language of constraint programming and uses a general purpose local search solver. Comet is now an abandoned project but the architecture used is still

of interest. The core of framework are the variables and invariants.

One layer above the invariants are the differentiable objects that can use the invariants and variables. Both constraints and objective are implemented as differentiable objects. They are called differentiable because it is possible to compute how the change of a variable value will affect the differentiable object's values. All constraints are implemented using the same interface, that means that all constraint have some methods in common. This is especially useful when combining multiple constraints in a constraint system, that also implements the constraint interface. The constraints can be combined in a constraint system that then uses the method from the individual constraints to calculated its own methods. Just like the constraint interface there exist a objective interface.

The next layer is where the user models their problem and use the objects mentioned above. Several search method are implemented that can be used. The benefit of this architecture is the user can focus on modeling the problem efficiently on a high level and thereby avoid small implementation mistakes. Using constraint programming inspired structure gives the benefit of brief but very descriptive code.
The idea of structuring the constraint and invariant as interface is also used in this framework.

3.4.4 OscaR

4 Architectural Overview

This section gives an overview of the key components of the solver. Figure 1 gives an overview of the most basic classes and the classes they have pointers to (access to).
The two engines for solving are the **GecodeEngine** and **LocalSearchEngine** that find the initial solution and optimize the solution respectively. **GecodeEngine** is used for preprocessing and finding an initial solution, if possible with in the limits given. **GecodeEngine** will be elaborated further in section 5.
LocalSearchEngine is responsible for the optimization part of the solver with the use of local search and metaheuristics. How the optimization is done is described in section 8. **LocalSearchEngine** transform the model to a model better suited for local search before the local search can start. How this is done and why will be discussed in section 6.
The **Storage** class contains pointers to components of a CBLS model, variables, constraints, and invariants. **LocalSearchEngine** can add new objects such as invariants to Storage.
Constraint and **Invariant** are super classes to all constraint and invariant respectively and are described in subsection 4.2.1 and 4.3.2. They contain abstract methods that the subclasses must specify.
The main part of the solver is the **General Solver** class that contains the engines used for solving. The **General Solver** class contains the methods that are called by the user, such as creating variables and constraints, finding initial solution and optimizing the solution and is described in subsection

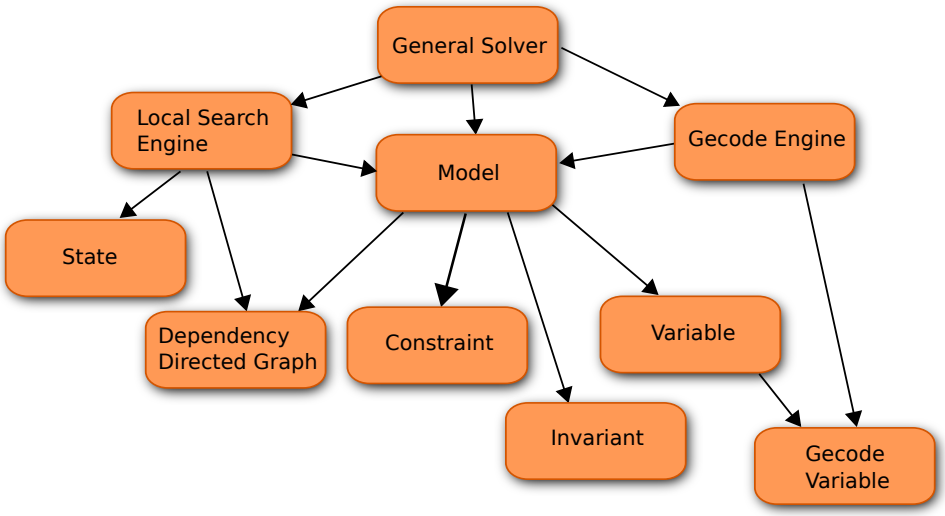

Figure 1: Overview of what the key classes of the framework contains pointers to

4.4.
The **Variable** class contains both the variable used by Gecode but is also used for local search.

4.1 Variables

The **Variable** only consist of short setters and getters methods. Each variable has a unique integer id in the order they are created. Their current value must be within their domain, that is given by a lower and upper bound. Each variable has a boolean that tell if the domain size of the variable is one and a boolean that tell if it is defined by a oneway constraint. They have a value for the number of constraints and oneway constraints they participate in.

4.2 Constraints

Constraints are all derived from the same class (**Constraint** figure 2) that forces some methods to be implemented. All constraints need a priority according to how important the constraint is. The priority is given by an positive integer and do not need to be unique but it will help the local search

Figure 2: Important methods in Constraint class

to differentiate between infeasible solutions. There is no reason to skip an integer in a sequence since it wont make a difference. It is suggested to keep the size of the sequence of priorities lower than 5.
An example of a constraint is the **Linear** constraint, which is the same as the one used in integer and binary programming, equation 4.2 is an example of a **Linear** constraint.

$$c_1 : \ 2x_1 + 2x_2 \leq 2 \qquad x_1, x_2 \in \{0, 1\} \tag{5}$$

A constraint is posted in the Gecode **Space** by **GecodeEngine** and later handled in the **LocalSearchEngine**. The constraints are treated differently in the environments and need different parameters and methods for that. The LS environment handles constraints through invariants hence an implementation of a constraint needs a method for creating the invariants needed in LS. The method createInvariants creates invariants that might be auxiliary variables helpful during local search and one invariant that represents if the constraint is violated and the violation degree. The violation degree can be one if violated and zero otherwise, or it give a measurement of violated the constraint is. For the linear constraint c_1 in equation 4.2 the violation degree is how much the value of the left hand side must decrease to satisfy the constraint. A helpful auxiliary variable is the value of the left hand side such that it does not need to be recomputed when computing the violation.
The methods canBeMadeOneway and makeOneway are used if the constraint can be transformed to a oneway constraint hence functionally define one of the variables. Method canBeMadeOneway returns a boolean whether it can be used or not and makeOneway transforms the constriant into a oneway constraint, implemented as an invariant, that defines one of the variables. The only constraint that has been implemented is **Linear**.

4.2.1 Implementation of Linear

Linear is a linear constraint c_j with a coefficient hashmap $A(c_j)$ and a vector of variables $X(c_j)$ that have some relation to a constant on the right hand side $b(c_j)$. The relation that can be used are: $\{\leq, =, \geq, >, <\}$ but only the first two $(\leq, =\})$ needs to be managed. If one of the three last is used it is transformed into a $\leq$ relation instead. This is done to reduce the number of comparisons later. The change can be done by multiplying with -1 on each side to change from "greater" to "less" relation. All coefficients and variables are integer hence all strictly less/greater relations can be transformed to a less/greater equal relation by increasing or decreasing $b(c_j)$ by 1.
The methods canBeMadeOneway and makeOneway are covered in section 6 and only **Linear** constraints that have "=" relation can be made oneway constraints.
The method createInvariants creates two invariants the first one represents

the value of the left hand side in the constraint and the other represents the violation degree. The first invariant is a **Sum** invariant and is described in the next subsection. The second one depends on the relation of the constraint and is either a **LEQVioaltion** or **EQVioaltion** that are also describe in the next subsection.
For notation if a variable is defined by an invariant it belongs to the set $Y(c_j)$ instead of $X(c_j)$. Algorithm 1 illustrates how the two invariants for **Linear** is created.

Algorithm 1: Linear - createInvariants()

input : **Constraint** c_j
output: two invariants

```
1  set invars = ∅
2  Sum y = Sum(A(c_j),X(c_j),Y(c_j))
3  int value = sumInvariant.setValue()
4  invars.add(sumInvariant)
5  if getPriority() ≠ 0 then
6      if relation is '≤' then
7          LEQviolation leq = LEQviolation(sumInvariant, b(c_j))
8          if value ≤ b(c_j) then
9              V(leq) = 0
10         else
11             V(leq) = (value − rightHandSide)
12         end
13         invars.add(leq)
14     else
15         EQviolation eq = EQviolation(sumInvariant, b(c_j))
16         if value == b(c_j) then
17             V(eq) = 0
18         else
19             V(eq) = |value − rightHandSide|)
20         end
21         invars.add(eq)
22     end
23 end
24 return invars
```

4.3 Invariants

Invariants are all derived from one common class **Invariant**, see figure 3, such that they all implements the same methods. This is very useful when doing local search. Invariants are only introduced after an initial solution to

an instance has been found and before the local search has begun. Invariants can represent either a variable or an auxiliary variable and are defined by oneway constraints. The invariants classes that are implemented contain information about how the invariant is defined and the value of the invariant. Hence the **invariant** classes are representing both the oneway constraint and the invariant.
All subclasses of **Invariant** has a delta value, a current value, coefficient map, a stack of changes, a lower bound, and a upper bound. These are used by the different methods of the invariants.
All Invariants must implement the methods proposeChange, calculateDelta , and updateValue that are used during local search. When suggesting a new value to a variable the method proposeChange is used. The proposed change is put on the stack until the the method calculateDelta is called. calculateDelta is by the delta evaluation function in local search and updates the delta value of the invariant according to the changes received from proposeChange. The method updateValue is called when a neighborhood operation is commited, to update the value of the invariant.
Each type of invariant must implement its own method since the methods can be different for each type of invariant.
Different classes uses the invariants but do not differentiate between them since they all have the same methods. If invariants did not have a common super class then each invariant type would need its own data structure for storage. Another benefit is the search procedures do not have to examine which invariants the model consist of since they all have the same methods. It also makes it easier to add new invariants since all the functionality are

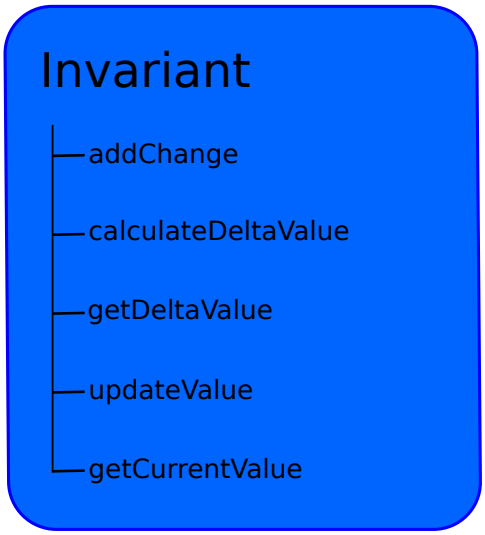

Figure 3: Important methods in Invariant class

implemented by the new invariant and nothing has to be changed in the **LocalSearchEngine**.

4.3.1 Implementation of Sum

The class **Sum** is used to define variables or auxiliary variables by a summation of variables multiplied by their coefficient with a constant offset. I.e. $x_1 = 2x_2 + 4x_3 + 1$, **Sum** can represent the right hand side in the equation defining x_1.
The method proposeChange(int $variableID$, int $changeInValue$) uses the variables id to get its coefficient from a hashmap. The coefficient multiplied by the integer $changeInValue$, the change of the variables value, gives the delta value that is pushed on a stack $variableChange$.
The method calculateDelta () first reset the delta value to zero and then popping each integer on the stack and add them to the delta value. The implementation can be seen in algorithm 2.

Algorithm 2: Sum - calculateDelta()

```
    input  : stack VariableChange
    output: bool allowed
 1  int DeltaValue = 0
 2  while VariableChange ≠ ∅ do
 3  |   DeltaValue += VariableChange.pop()
 4  end
 5  if DeltaValue + CurrentValue < lowerbound then
 6  |   return false
 7  else if DeltaValue + CurrentValue > upperbound then
 8  |   return false
 9  else
10  |   return true
11  end
```

It checks if the new value would violated the bounds of the **Invariant** in case it is used to define a variable. By returning false it tells the new value would not be within the bounds of the invariant hence the change cannot be allowed. This will be used during local search in section 8.
It updates the by adding the delta value to the current value in the method updateValue().

4.3.2 Implementation of LEQViolation and EQViolation

The invariant **LEQViolation** and **LEQViolation** are used for measuring violation of a constraint. They are a relation between the value of an invariant and an integer. The value of **LEQViolation** is zero if and only if

the value of the invariant is less than the integer. The value is the difference between the invariants value and the integer otherwise. For **LEQViolation** the value is zero if they are equal value otherwise the absolute value of the difference.
proposeChange and updateValue are implemented almost same as in **Sum** but proposeChange does not use the coefficient hashmap.
Algorithm 3 and 4 is the implementation of calculateDelta .

Algorithm 3: LEQViolation - calculateDelta()

input : stack VariableChange, int LHS
output: **bool** allowed

```
1  if VariableChange = ∅ then
2  |  DeltaValue = 0
3  |  return true
4  end
5  if LHS + VariableChange.pop() ≤ RHS then
6  |  DeltaValue = −CurrentValue
7  else
8  |  int old = max(LHS − RHS, 0)
9  |  int new = max(LHS + VariableChange.pop() − RHS, 0)
10 |  DelataValue = new − old
11 end
12 return true
```

Algorithm 4: EQViolation - calculateDelta()

input : stack VariableChange, int LHS
output: **bool** allowed

```
1  if VariableChange = ∅ then
2  |  DeltaValue = 0
3  |  return true
4  end
5  if LHS + VariableChange.pop() ≤ RHS then
6  |  DeltaValue = −CurrentValue
7  else
8  |  int old = max(LHS − RHS, 0)
9  |  int new = max(LHS + VariableChange.pop() − RHS, 0)
10 |  DelataValue = new − old
11 end
12 return true
```

The bounds of these invariants are always satisfied.

4.4 General Purpose Solver - GPSolver

The **GPSolver** class contains the most high level methods and most of them are used directly by the user. An overview of the most important methods is shown in figure 4.

The method `createVariables` takes three arguments to create a number of variables. The first argument is the number of variables to create with the given lower and upper bound, the second and third argument respectively. The method creates both the gecode variables used in the **GecodeEngine** and variables used in the **LocalSearchEngine** class. The variables used in the **LocalSearchEngine** class (LS variables) are of the class **Variable** and each has a pointer to the associated Gecode variable. The method returns pointers to the LS variables.

The method `relax` is only used if an initial solution could not be found within the limits given. It controls how the instance should be relaxed to find an initial solution. There is currently only one method implemented that relaxes the an instance that is described in subsection 5.2.

All constraint available are created by calling the associated method in **GPSolver** , that calls the constructor of the constraint and the method in **GecodeEngine** for posting the constraint in a Gecode space. The constraint objects should not be created directly by the user since the Gecode **Space** class is not available to the user.

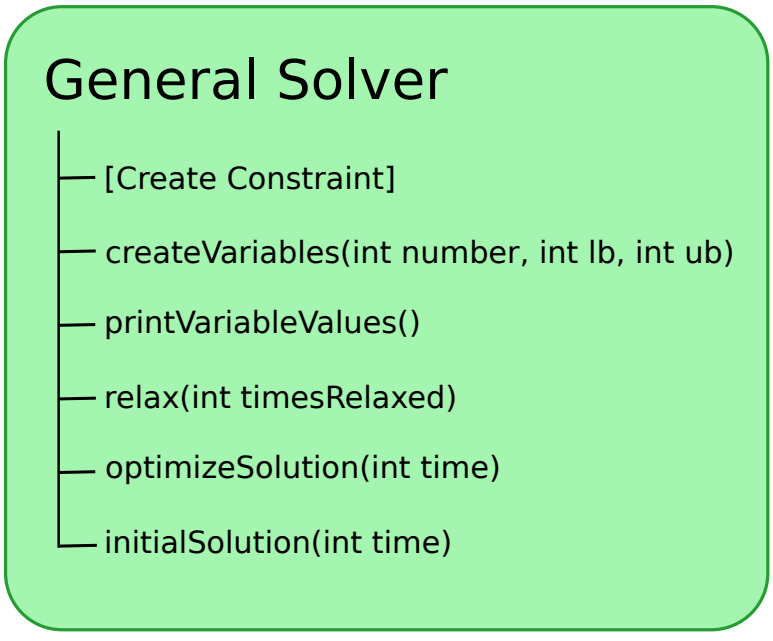

Figure 4: Important methods in GPSolver class

To implements a new constraint object it must inheriting from the **Constraint** class and two methods, one in **GPSolver** and one in **GecodeEngine**, must be implemented. The method in **GecodeEngine** must post the constraint in the **GecodeEngine** space. The method in **GPSolver** should call the constructor of the constraint implemented and call the method in **GecodeEngine**. The solver must be able to reproduce the call to **GecodeEngine** in case it an initial solution is not found within the limits given. The relaxation method must be updated to handle the new constraint implemented as well.
To find an initial solution the method `initialSolution` must be called and it takes an integer argument. The argument indicates the time Gecode is allowed to search for an initial solution before `relax` is called. Once `relax` is called the same time limit is given again.
To find a better solution than the initial solution the method `optimizeSolution` can be called with a time limit as argument. That method starts the local search that is described in section 8.

5 Preprocessing and Initial Solution

All constraints that are created are both saved in **Storage** and parsed to **GecodeSolver**, that is derived from Gecode Space. In **GecodeSolver** they are posted such that Gecode can be used to find an initial solution. Gecode has a large selection of constraints that can be used [7, p. 58-80]. Most of the constructors of these constraints are overloaded such that they can take different arguments and works with different types of variables.

5.1 Domain Reduction

When an initial solution to the model is requested, Gecode Branch() method is called that specifies which variables to branch on and what values should be examined in the branches. Gecode uses preprocessing before searching for a solution and that might reduce the domain of some of the variables. If the domain of a variable is reduced to a single value, the variable is said to be fixed and is assigned that value. The variable will never change value hence it is no longer considered an independent variable.

5.2 Finding an Initial Solution

Once domain reduction preprocessing has been done a Gecode DFS search engine is started. The stop criteria for Gecodes search can be specified by an option class. A Gecode search engine takes a space and search option as arguments and the search option contains a stop object. The stop object can either be timestop, nodestop or failstop. Each time Gecode branches on a variable two new nodes are created and nodestop set a upper bound on the number of nodes to explorer. If Gecode reaches a node that has no feasible assignment of one or more variables then that space is failed and failstop sets an upper bound on the number of failed spaces that can occur before stopping. Timestop stops the search if the time limit is reach.
Instead of using only one of these stop object a **Multistop** object is created that combines all three stop objects such that it can have multiple stopping criteria.
Combinatorial problems can be formulated with Gecode and these problems can be very difficult to solve. In these cases Gecode keeps searching for a solution until it finds one (or runs out of available memory). Instead, the search can be stopped using stop object and the constraints can be relaxed such that Gecode can find a initial solution to the instance. If one of the stop criteria is reached the relax is called and some of the constraint are relaxed. To relax some constraints a new Gecode Engine is created and all variables and constraints, except those relaxed, are created and added to the new space. In order to choose which constraint that should be removed the priority of the constraints, given by the user, is used. The constraint

is only relaxed if all non functional constraints with lower priority has been relaxed. The functional constraints are the last to be relaxed no matter what their priority is. The reason for this choice is these constraints are used to create oneway constraints and are only created if the functional constraints is feasible. This will be discussed further in the next section.
A greedy approach is chosen in order to keep the time usage low. Each time Gecode fails in finding a solution the number of constraints added next time is halved. This is repeated at most 2 times, down to posting 25 % of the constraints. If no solution can be found within the search limits, the search is stopped and finding an initial solution to the instance has failed.
The constraints are chosen by their priority and ties are broken at random. I.e. a model with 100 constraint of priority 1, 40 of priority 2, and 15 of priority 3 and there are 20 constraints that should be relaxed. The 15 constraints with priority 3 and 5 of the 40 constraints with priority 2 are chosen at random would not be posted. The constraints that are not posted in the Gecode space are still applied when doing local search, hence the initial solution might start with some violations.
If Gecode fails to find an initial solution, the independent variables are assigned value within their domain chosen uniformly at random.

6 Structuring Local Search Model

Once an initial solution to the constraint satisfaction problem (CSP) has been found by Gecode the model is transformed to create a model suited for local search, a CBLS model. Two new datastructures structures are introduced in this section, dependency directed graph in subsection 6.2 and propagation queue in subsection 6.3.
The dependency directed graph is used to update invariants when a variable changes value. A propagation queue q_i is created for each variable x_i that gives an ordering of the invariants reachable from x_i in the dependency directed graph.
The model is simplified by defining some of the variables by transforming the functional constraints into oneway constraints using the algorithms implemented in the respective constraints. When a variable is defined by a oneway constraint it is transformed into an invariant since its value is dependent on other variables and invariants.
The only constraint implemented, **Linear**, is used as an example for creating oneway constraints.

6.1 Simplification

Each functional constraint can be used to define a variable making it dependent on other variables or invariants. The idea is to reduce the number of independent variables reducing the search space and likely the neighborhoods in local search. This does come with a downside that calculation of a variable changing value might take more time.
The functional constraint used to create a oneway constraint must be satisfied by the initial solution in order to create a oneway constraint. Once a oneway constraint is made it defines a variable that is represented by an invariant. The invariants value is always within the domain of the variable which corresponds to the functional constraint always being satisfied.
Even though all **Linear** constraints with an equality relation are functional only those with unit coefficients only are chosen to be functional for simplicity. I.e. on the form $c : \sum_{i=1}^{\alpha(c)} a_i x_i = b(c),\ |a_i| = 1\ \ \forall i$. If other coefficients where allowed it could create non integer coefficients when defining a variable, and does not work with the rest of the framework currently.
For each functional **Linear** constraint c_j with unit coefficient two algorithm steps are used to create invariants. The first checks if the constraint c_j can be transformed into a oneway constraint and the other transforms c_j into a one-way constraint defining x_i.

Algorithm 5: Linear - canBeMadeOneway()

```
1  Variable bestVariable = NULL
2  int numberOfTies = 0
3  // Break ties
4  if defines(x_i) < defines(bestVariable) then
5  |  // Choose the variable that helps define fewest
   |     invariants
6  |  bestVariable = x_i
7  |  numberOfTies = 0
8  end
9  else if defines(x_i) == defines(bestVariable) then
10 |  if |deg(x_i)| < |deg(bestVariable)| then
11 |  |  // Choose the variable with lowest degree
12 |  |  (remember to define degree) bestVariable = x_i
13 |  |  numberOfTies = 0
14 |  end
15 |  else if |deg(x_i)| == |deg(bestVariable)| then
16 |  |  // Fair random
17 |  |  numberOfTies++
18 |  |  if Random(0,numberOfTies) == 0 then
19 |  |  |  bestVariable = x_i
20 |  |  end
21 |  end
22 end
23 if  bestVariable ≠ NULL then
24 |  makeOneway(Variable bestVariable)
25 end
```

For each unit **Linear** constraint an independent variable is found if possible. If there is more than one eligible variable the best variable among those is found. The first tiebreaker is the number of oneway constraints the variable participate in (helps define other variables). The next tiebreaker is the number of constraints the variables participate in. If none of the tiebreakers can be used a fair random is used such that the probability is equal for all variables whose ties could not be broken.
Once a the best variable is found, if any, the algorithm 6 `makeOneway` is called.

Algorithm 6: Linear - makeOneway(Variable x_i)

input : Variable x_i
output: A new invariant y defined by a oneway constraint

```
1  set Q = ∅                                   // new coefficient set
2  set U = ∅                                   // new variable set
3  // Move x_i to right hand side and set coefficient to 1
4  foreach x_k in X(c_j) \ x_i do
5  |   c'_kj = -c_kj / c_ij
6  |   Q = Q ∪ c'_kj
7  |   U = Q ∪ x_k
8  end
9  // Move right hand side to left hand side
10 int b' = b(c_j) / c_ij
11 invariant y = Sum (U,Q,b')
12 // Invariant whose value is defined by the other
      variables and a constant
```

The algorithm transforms the constraint c_j into a oneway constraint defining an invariant. The dependency directed graph G is updated by adding the new invariant inv and removing the constraint c_j and variable x_i.

6.2 Dependency Digraph

For each constraint not transformed to a oneway constraint auxiliary variables are introduced as invariants and an invariant for violation is created. The are created by calling the method createInvariants of each constraint. The algorithm for **Linear** was shown in subsection 4.2.1. These invariants are used to speed up local search, that is described in section 8. **(Are they?)**

The dependency directed graph (DDG) $G = (V, A)$ is made of a set of vertices V representing all independent variables and all invariants. To ease the notation we say the value of vertex $v \in V$ is the value of the variable or invariant it represents.

The vertex $v \in V$ has an outgoing arc to vertex $u \in V$ if and only if the value of u is directly dependent on the value of v. The vertices representing independent variables never have an ingoing arc.

The graph can be illustrated with all the variable vertices to the left with outgoing arcs going right to vertices representing invariants.

If a variable is defined by a oneway constraint the variable vertex is removed from G since the value of that variable is given by the invariant representing it.

The DDG is used to update values of variables and invariants during local

search. The graph G is also used to build the propagation queues described in subsection 6.3. The idea of a graph representing the relationship between invariants comes from Comet [8, p. 97] and OscaR[2, p. 7-9].
To illustrate how the DDG is made an example of a model with three variables and a two constraint will illustrate it.

$$\begin{aligned} c_1: &\quad x_1 + x_2 - x_3 &&= 1 \\ c_2: &\quad 2x_2 + x_3 &&\leq 2 \end{aligned}$$

The variable x_3 can be defined as an invariant y_1 by transforming c_1 to a oneway constraint $c_1' : x_3 = x_1 + x_2 - 1$. Once variable x_3 is defined by a oneway constraint x_3 are removed from the graph and replaced by invariant y_1. The variables x_1 and x_2 defines y_1 hence they have outgoing arcs to y_1. Auxiliary variable can be useful to update constraint violations and in this example we could create an auxiliary variable where value is the sum of the left hand side of c_2. The auxiliary variable will be represented by a **Sum** invariant y_2 which will be added to G. The invariant y_1, representing x_3, and variable x_2 have an outgoing arc to y_2 that has an outgoing arc to c_2. A **LEQViolation** invariant y_3 representing the violation of constraint c_2 is added as well. Invariant y_1 has an outgoing arc to y_2 since Variable x_3 participates in c_2.
When changing the value of x_2 both invariants need to be updated since they are dependent on the value of x_2. Invariant y_2 is dependent on the value of y_1 therefore to avoid updating y_2 twice, it is beneficial to update y_1 before updating y_2. This is the ordering given the propagation queue that is discussed in the next subsection.

In order to avoid circular definitions of invariants dependency directed graph G is made acyclic. A circular definition could be if x_i is used to define x_j and vise versa. Then a change in value of x_i would lead to a change in value of x_j that again changes the value of x_i and so on.
In order to remove circular definitions, all strongly connected components of size two or more in G is found. A *strongly connected component* (SCC) is a maximal set of vertices V^{SCC} such that for each pair of vertices $(u, v) \in$

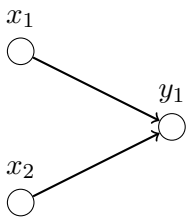

Figure 5: Small example of DDG

V^{SCC} there exist both a path from u to v and a path from v to u [3, p. 1170]. Each of these strongly connected components must be broken in order to keep G acyclic, since a SCC consist of at least one cycle. A SCC can be broken by removing arcs and/or removing vertices. The arcs A represent relations between variables, invariants and constraints and should not be changed. The vertices V^{SCC} can only be vertices representing invariant since variable only have outgoing arcs and constraint only have ingoing arcs. Undefining one of those invariants corresponds to removing one of the vertices, hence breaking the SCC. An invariant can be undefined by reintroducing the variable it represents and removing the invariant from the model. The oneway constraint used to define the invariant is transformed back into a functional constraint again and is reintroduced in the model.
For each SCC one of the vertices is chosen and the invariant it represents is undefined. The invariant is chosen in the order of lowest domain then highest arity of oneway constraint defining it. If there is ties they are broken at random.
To find all SCC of size two or more Tarjan's algorithm for finding strongly connected components is used. **(cite SIAM J. Comput., 1(2), 146–160.)**. Tarjan's algorithm has a depth first search behavior when exploring vertices and uses a stack when exploring. The stack consist of vertices that has been visited before but not part of a strongly connected component yet. The use two counters, first one index is unique in the order they are visited the other is lowlink initially the same as index. If we go from vertex v to vertex u and u, $v.lowlink = min(v.index, u.lowlink$. When a vertex does not point to any more vertices if its index is equal to it lowlink it pops the stack until it reaches it self. All the vertices popped from the stack is in a strongly connected component.
The algorithm has been modified to give each variable a time stamp when they are popped from the stack aswell. That time stamp gives an ordering of the invariants used to create the propagation queues described in the next subsection. The time stamp is updated each time the algorithm is repeated, when cycles are found.
Removing the fewest possible invariants such that there are no cycles in G

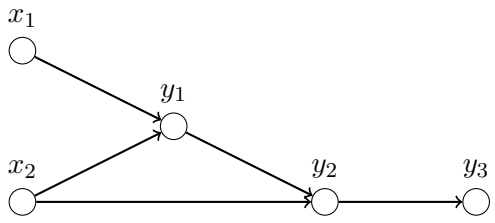

Figure 6: Small example of DDG continued

corresponds to solving the minimum feedback arc set problem and is known to be NP-hard [2, p.9]. In order to reduce the construction time a greedy approach is chosen.

Once these strongly connected components are broken there is still no guarantee that G is a directed acyclic graph (DAG). A strongly connected component can be made of several cycles and it might not be sufficient just to break all SCC found by Tarjans algorithm initially. The process of finding SCC with Tarjans algorithm and then breaking these strongly connected components is repeated until no strongly connected components (of size two or more) is found by Tarjans algorithm.

The only vertices that can make strongly connected components are the vertices representing invariants, more specifically only the invariants that define a variable. Therefore the check for strongly connected components are done before other invariants are added.
The first tiebreaker in algorithm 6 is used as a heuristic to reduce the number of cycles generated. If invariants are not used to define other invariants no cycles can occur, since cycles can only be made of invariant vertices.

6.3 Propagation Queue

For each independent variable x_i a propagation queue q_i is made. A propagation queue q_i is an topological sorting of invariants that are reachable from the vertex representing x_i in the dependency directed graph G **(maybe define topo sort)**. The propagation queue q_i is used such that each invariant dependent on the value of x_i is updated at most once if the variable changes value. The DDG represents which invariant that are directly affected by a change in variable x_i but not the order in which they should be updated. Figure 6.3 shows the necessity of such an ordering.
If inv_1 is updated before inv_2 then it might need to be updated again after inv_2 is updated hence updated twice. In worst case the number of updates performed when updating x_i could be exponential in the number of vertices reachable from x_i instead of linear.
Once the dependency directed graph is a DAG each invariant vertex has

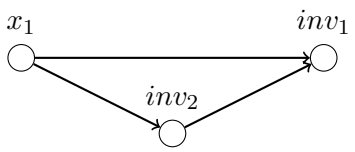

Figure 7: Importance of propagation queue

been given a time stamp by Tarjans algorithm. Propagation queues are implemented as red-black trees without duplicates hence they have insert time complexity $O(log(n))$. For each variable vertex x_i in dependency digraph G a depth first search is made and each vertex visited is added to the propagation queue of x_i **(currently revisits vertices and adding vertices pointed to agian. Should be fixed)**. The vertices in the propagation queue are ordered according to their time stamp in decreasing order which is a topological sorting such that there is no backward pointing arc.

During local search when a single variable x_i changes value the change propagate through the DDG using the ordering from the propagation queue q_i. When two or more variable change value the propagation queues are merged into a single queue removing duplicates.

7 From Modeling to Local Search

Simple problem will be modeled in this section and the different parts of this constraint based local search solver will be illustrated.
(New subsection- The problem)
A city have budget of 50 million kr. for a festival and have some suggested activities. The activities have an individual cost and have been given a score based on how successful the activity is expected to be. The duration of the festival is three weeks and the activities have individual requirement to which week the can be planned. Exactly two must be planned the first week, at most two the second week and at most one the third week. The job is to plan activities with in the budget and schedule maximizing the score. Table 1 shows the data given.
If an activity is chosen it is chosen for all the weeks with yes.
(Section - Model) Each of the activities can be represented by a binary variable whether the activity is chosen or not. The constraint of each week can be modeled with the **Linear** constraint **(Somewhere I should describe the constraints and invariants implemented)**.

Activity No.	Cost	Score	Week 1	Week 2	Week 3
1	20	3	yes	yes	no
2	10	2	yes	no	no
3	35	4	no	no	yes
4	42	5	yes	yes	yes
5	13	2	no	yes	no
6	37	4	yes	no	yes

Table 1: Suggested activities

8 Local Search and Metaheuristics

Important: Neighborhoods, FLip and swap?. Search methods RW, BI, and FI (sidewalks). Heuristics obj var only and conflicting var only. Meta heuristic, Tabu search (tabu tenure, aspiration, Search procedure FI or BI, Neighborhood limitations for large neighborhoods?). Combining search procedure. **(Make overview of classes and methods)**

Local search can be seen as a walk through the neighborhood graph **(Should be defined earlier)** going from one sol

The **LocalSearchEngine** class uses the model created for local search described in section 6 and uses local search to improve the initial solution. When talking about variables in this section, it only refers to independent variables unless otherwise stated. Pointers to the independent variables are kept in a standard vector, called mask, and is shuffled to give a random sequence.

Local search explores how changing value of few variables will affect the solution quality, hence exploring a neighbor solution. The key for being efficient is to compute this evaluation fast and the dependency digraph and the propagation queues are used for this. For simplicity let us first consider the step involving changing the value of a single variable x. The change of variable value is send through the dependency digraph where each invariants, reachable from the vertex representing x, is updated. The propagation queue is used to determine the sequence the invariants should be updated. This will update invariants that represent violation of constraints with different priority and the evaluation function.

If a step consist of more variables changing values, such as swapping values of two variables, the propagation queues of these variables can be merged. The merging should remove duplicates and keep the invariants topological sorted. The invariants unique time stamp can be used to keep the ordering. The new queue gives an ordering the invariants should be updated such that they are only updated once during each step.

Before making a step several neighbor solution might be explored before choosing a neighbor solution by applying a neighborhood operation. To evaluate a neighborhood operation a delta value for each invariant is used. The delta value is the value an invariant would change if the neighborhood operation is performed. By this we can evaluate the neighbor solution without performing a step.

Each constraint $c \in C$ created in the model was given a priority p and these priorities are used during local search and let k denote the highest priority given. **(Define $V(c)$, violation degree and $P(c) > 0$ priority of c)**. Let $q_p \in Q$ be the sum of violation degree $V(c)$ of constraints with priority p. The evaluation functions value is consider as q_0 and the vector Q is used to evaluate the quality of a solution. Two candidate solutions τ and τ' each

have a vector of quality Q_τ and $Q_{\tau'}$ respectively. To determine which of the two solution are best their vector can be compared, starting with position k and going backwards. The first position they have different value determine which solution is best, the one with the lowest value. Illustrated with a small example:

$$Q_\tau = (5, 2, 4, 2) \tag{6}$$
$$Q_{\tau'} = (10, 6, 3, 2) \tag{7}$$

Violation degrees of constraints with priority 3 ($Q_\tau[3]$, $Q_{\tau'}[3]$) contributes with 2 in each candidate solution and then the violation degree of priority 2 is consider. Then candidate solution τ' is consider better than τ since $Q_\tau(2) = 4 > Q_{\tau'}(2) = 3$. An invariant is created for each priority p and one for the evaluation function before the local search begin at the same time invariants are created by each constraint class.
The new classes used for local search are in three different categorize, moves, neighborhoods, and search procedures. A **Move** object stores information of a neighborhood operation including the change of the evaluate function and change of violations. A subclass of the **Neighborhood** class is the choice of step function and gives the sequence the neighbor solutions are explored. They also determine how neighborhood operation are calculated and how steps are performed. The search procedures can quire a **Neighborhood** class to evaluate a neighborhood operations, a **Move**, effect on the evaluation function. It is the search procedures that determine which neighbor solution to go to if any. Neighborhoods and search procedures are combined to create different local search algorithms and **LocalSearchEngine** uses them within the time limit to improve the solution.

8.1 Neighborhoods

The neighborhood classes are all subclasses of the super class **Neighborhood** such that they can easily be combined with the search procedures. The methods of neighborhoods are illustrated in figure **??**.

All **Neighborhoods** implemented uses a step functions that changes value of a single independent variable, from 0 to 1 or vise versa since all variables are binary. A neighborhood operation is stored in an a **Move** that contains a pointer to the variable used, the variables change in value, and the change to the quality vector Q, once computed. The change in the quality vector is referred to at the *delta vector*.
The **Neighborhood** classes that are implemented are shown in table 2.

The **RestrictedFlipNE** class chooses a variable for the next move with probability $p = \frac{5000}{n}$, where n is the number of independent variables. This

gives expected 5000 variable before it return a **NULL** pointer.

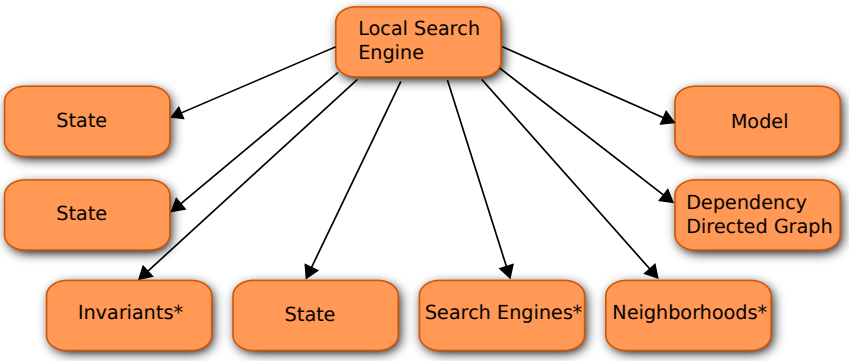

Figure 8: Overview of the class pointers of Local Search Engine. The fields marked with a star (*) are several classes of that type.
(Variable og constraint til venstre, think this should be posted earlier)

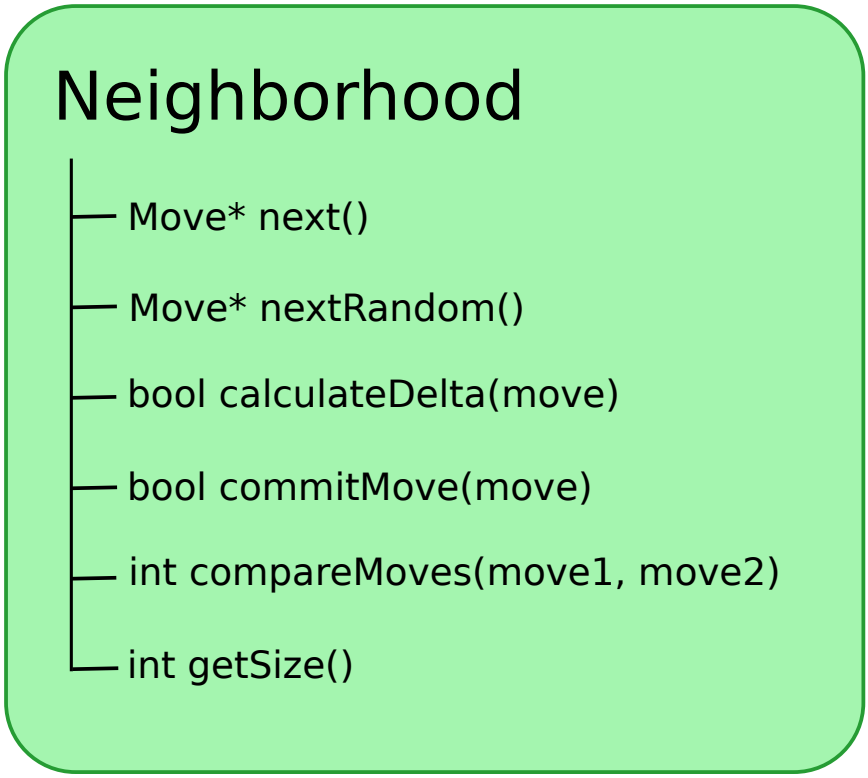

Figure 9: The methods all Neighborhood classes needs to implement.

class	Heuristic
FlipNeighborhood	All variables
RestrictedFlipNE	A expected 5000 variables chosen at random
ConflictOnlyNE	All variables in unsatisfied constraints
RandomConflictFlipNE	Variables from a random unsatisfied constraint

Table 2: Table of **Neighborhood** classes

The method next() creates new **Move** object and returns a pointer to it. If all the different neighborhood operation has been returned the method returns a pointer to **NULL** instead. To know when a neighborhood has been fully explored, counters and iterators are used depending on the neighborhood. These are reseted when returning **NULL** or the method commitMove(move) is called. The **Move** created only contain the variable and its suggested change in value, the delta vector is not computed yet.
Method nextRandom() gives a random neighborhood operation without changing the.
Method calculateDelta (move) takes a **Move** pointer as argument and propagate the change through the dependency digraph using the propagation queue of the variable. The method identical for all the **Neighborhood** classes implemented. A neighborhood operation that calculate the delta change if variable x_i would change value is describe with the following step:

1. Reset delta value of invariants in quality vector Q
2. Send delta value of x_i to neighbor invariant in DDG.
3. For each invariant inv in propagation queue of x_i, calculate invs delta value, if it is not zero, send it to neighbor invariants in DDG.
4. If a variables delta value is not allowed, by a oneway constraint, reset all delta values of invariant in the propagation queue.
5. Otherwise set $move$s delta quality vector.
6. return if the $move$ is an allowed neighborhood operation.

The delta values of the invariants can be reset by calculating delta value, when no change is send. The reason for resetting them is only to make sure their queue of changes is empty before the next neighborhood operation.
To perform a step the method commitMove(move) is called with a pointer to the **Move** that used be used. commitMove(move) use the delta value calculated by calculateDelta (move) to update the value of invariants. The delta value needs to be recomputed since other neighborhood operation might have been suggested. Once the delta values of invariants have been computed they can be added to their current value. Invariants that represent

violation of a single constraint are kept in a hash map of they are non zero. If they change value from non zero to zero or vise versa, that hash map needs to be updated. The hash map is used by the two neighborhoods **ConflictOnlyNE** and **RandomConflictFlipNE** that only can be used when the current solution is infeasible.
A default method compareMoves(move1,move2) compares the delta vector of two **Move** pointers and returns 0 if they are the same, 1 if if move1 is best and 2 otherwise.
The size of the neighborhood and the restriction applied to it, if any, can be requested from the method getSize(). It returns the size of the current neighborhood, for **ConflictOnlyNE** and **RandomConflictFlipNE** the neighborhood size can change after each step, for the others it is a constant size.

8.2 Search Procedures

Neighborhood classes do not implement any strategy of which neighborhood operation that should be chosen. Search procedures are using a neighborhood and define this strategy. The classes implemented are **FirstImprovement**, **BestImprovement**, **TabuSearch**, and **RandomWalk**. **FirstImprovement**,
BestImprovement, and **RandomWalk** are implementation of local search algorithms of almost same name and can be used together with any of the **Neighborhood** classes. **TabuSearch** is an implementation of the metaheuristic tabu search using best improvement, a tabu tenure, and an aspiration criteria.

The class **BestImprovement** looks at each **Move** a **Neighborhood** class NE gives and finds the best **Move**. The best **Move** is determined from their delta vector after the method claculatDelta of the **Neighborhood** is called on each **Move**. It returns a boolean that tells if the current solution was improved. A boolean can be given to **BestImprovement** that indicate if it should commit a non improving **Move**. How each iteration is done is describe by algorithm 7

Algorithm 7: BestImprovement Start

```
input  : bool alwaysCommit
output: bool improvement
Move bestMove = NE.next()
Move move = NE.next()
while move != NULL do
    bool allowed = NE.calculateDelta(move)
    if !allowed then
        move = NE.next()
        continue
    end
    bestMove = compareMove(move,bestMove)
    move = NE.next()
end
bool improvement = Check if bestMove gives improvement
// by looking at delta vector

if improvement or alwaysCommit then
    NE.commitMove(bestMove)
end
return improvement
```

If **BestImprovement** is combined with the neighborhood class **RandomConflictConNE** it gives a minimim conflict heuristic that can be useful to reach a feasible solution.

FirstImprovement has a very similar implementation. Instead of calculating each **Move** of a **Neighborhood** class NE it stops requesting a **Move** once an improving **Move** is found. If no improving **Move** is found when NE returns a **NULL** pointer it does not commit a **Move**. If no improving **Move** is found the current solution is in a local optima with the regard to the chosen **Neighborhood**.

The class **RandomWalk** uses the method nextRandom() from its **Neighborhood** NE and if that **Move** is allowed it is committed. It takes an integer as argument that indicate the number of times it is repeated. The benefits is create new solution very fast but they are not likely to have a good quality. Though tabu search is a metaheuristic it is implemented the same way as the other search procedures but with some additions. It takes four arguments; the number of steps made(iterations), the best solution found, the current solution, and a tabu list. The implementation is similar to **BestImprovement** with additional checks with regard to the tabu list and aspiration criteria. If a neighborhood operation is tabu but leads to a solution better than one found so far, the tabu list is ignored. The algorithm for tabu search for a single flip neighborhood sketched by algorithm 8.

Algorithm 8: TabuSearch Start(iteration, best,current,tabulist)

```
input : int iteration, int[] best, int[] current, int[] tabulist
int tabuTenure = Random(0,10)+ min(NE.getSize()*2,
tabulist.size() /200)
Move bestMove = NE.next()
Move move = NE.next()
while move != NULL do
    bool allowed = NE.calculateDelta(move)
    if !allowed then
        move = NE.next()
        continue
    end
    bool isTabu = (iteration - tabulist[move.ID]) <= tabutenure
    if isTabu then
        if betterThanBest(current,move.getDeltaVector(), best)
        then
            NE.commitMove(move) tabulist[move.ID] = iteration
            return true
        end
        move = NE.next()
        continue
    end
    bestMove = compareMove(move,bestMove)
    move = NE.next()
end
NE.commitMove(bestMove)
```

TabuSearch needs to be combined with a **Neighborhood** class that uses single flip neighborhood operation. This makes it less flexible in combining it with a **Neighborhood** class than the other search procedures **FirstImprovement**, **BestImprovement**, and **RandomWalk**.
In order to create an efficient local search we need to change search procedure at some point, with the exception of tabu search that can perform well on it own.

8.3 Local Search Algorithms

When a model better suited for local search has been made the remaining time before the time limit is used to do local search. The check is done after a step of a search procedure has been made to insure the time limit is not exceeded. The best solution found while searching is save in a **State** such that the search can continue but always report the best solution when the time limit is reached.

Three algorithms has been made from combining **Neighborhood** classes and search procedures that will be used to test efficiency of the framework. The first algorithm uses two **TabuSearch** with different **Neighborhood** classes. When the solution is infeasible **TabuSearch** is combined with **ConflictOnlyNE** to only look at variables that can reduce the number of violations. When the current solution is feasible **TabuSearch** with a **RestrictedFlipNE** class is used. If the number of independent variables are less than or equal to 5000 it uses a **FlipNeighborhood** class instead. The reason for choosing a subset of the neighborhood to examine is to increase the number of steps made in case the neighborhood is large.

Algorithm 9: Local Search - Test Algorithm 1

```
1  ConflictOnlyNE neigborhood
2  RestrictedFlipNE neigborhood2
3  TabuSearch TSCON(neighborhood)
4  TabuSearch TSRFN(neighborhood2)
5  int iteration = 0
6  int [] best = getSolution()
7  int [] current = getSolution()
8  int [] tabulist(neighborhood.getSize(), -neighborhood.getSize())
9  while within time limit do
10 |  if Current solution is feasible then
11 |  |  TSCON.start(iteration ,current, best, tabulist)
12 |  |  iterations++
13 |  |  if getSolution() is better than bestSolution then
14 |  |  |  bestSolution = getSolution()
15 |  |  end
16 |  else
17 |  |  TSRFN.start(iteration, current, best, tabulist)
18 |  |  iterations++
19 |  |  if getSolution() is better than bestSolution then
20 |  |  |  bestSolution = getSolution()
21 |  |  end
22 |  end
23 end
```

The second algorithm for testing is iterative improvement using first improvement and random walk with a single flip neighborhood. They idea is to find a local optima fast, and use randomness to escape the optima.

Algorithm 10: Local Search - Test Algorithm 2

```
1  FlipNeighborhood FN
2  int randomMoves = min(FN.getSize() / 50, 10)
3  FirstImprovement FI(FN)
4  RandomWalk RW(FN, randomMoves)
5  bool improvement = true
6  while within time limit do
7  |  while improvement AND within time limit do
8  |  |  improvement = FI.start()
9  |  |  iterations++
10 |  |  if getSolution() is better than bestSolution then
11 |  |  |  bestSolution = getSolution()
12 |  |  end
13 |  end
14 |  RW.start()
15 |  iterations += randomMoves
16 |  if getSolution() is better than bestSolution then
17 |  |  bestSolution = getSolution()
18 |  end
19 end
```

The last algorithm that will be tested uses a minimum conflict heuristic with a single flip neighborhood when the current solutions is infeasible. When the current solution is feasible a tabu search with a restricted single flip neighborhood is used.

Algorithm 11: Local Search - Test Algorithm 3

```
1  RandomConflictConNE neigborhood
2  RestrictedFlipNE neigborhood2
3  BestImprovement BIRCC(neighborhood)
4  TabuSearch TSRFN(neighborhood2)
5  int iteration = 0
6  int [] best = getSolution()
7  int [] current = getSolution()
8  int [] tabulist(neighborhood.getSize(), -neighborhood.getSize())
9  while within time limit do
10 |  if Current solution is feasible then
11 |  |  BIRCC.start()
12 |  |  iterations++
13 |  |  if getSolution() is better than bestSolution then
14 |  |  |  bestSolution = getSolution()
15 |  |  end
16 |  else
17 |  |  TSRFN.start(iteration, current, best, tabulist)
18 |  |  iterations++
19 |  |  if getSolution() is better than bestSolution then
20 |  |  |  bestSolution = getSolution()
21 |  |  end
22 |  end
23 end
```

Several other algorithms can be made from the **Neighborhood** and search procedure classes. Though they can be combined in many ways they are not as easily combined as wanted. The **Neighborhood** classes can be a mix of heuristics and a neighborhood which is not ideal. The next subsection will discuss this in more detail.

8.4 Change in Local Search Design

9 Tests

9.1 Gecode Search Engine

10 Results

11 Future Work

(Short description of problems that should be investigated more)
(preprocessing :) preprocessing of Cplex and gurobi with Gecode
(Cycles :) Finding all (elementary) cycles in the dependency digraph

and/or finding the smallest set of vertices to remove such that the graph is DAG.
(Integer variables :) How to treat integer variables when they cannot be defined by oneway constraints.
(Propagation queue :) Finding a datastructure better suited for propagation queues than red-black trees (C++ set).
(Mixed neighborhood :) Find a way to treat both integer and binary variables in local search.
(Make "true" DFS when creating propagation queues, no revisit of vertices)
(Change Constraints to invariants during local search to measure the violation)
(option class for user to specify stop limits, branch other things)

12 Conclusion

References

[1] Handbook of constraint programming. Foundations of Artificial Intelligence, chapter 10, pages 266–290. Elsevier Science, 2006.

[2] A constraint-based local search backend for minizinc. 2015.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Jiri Matousek and Bernd Gärtner. *Understanding and Using Linear Programming (Universitext)*. Springer, 2006.

[5] Wil Michiels, Emile Aarts, and Jan Korst. *Theoretical Aspects of Local Search*. Springer, 2007.

[6] Sophie Demassey Nicolas Beldiceanu. Global constraint catalog. `http://sofdem.github.io/gccat/`. [Online; accessed 17-12-2015].

[7] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling part. In *Modeling and Programming with Gecode*. 2015. Corresponds to Gecode 4.4.0.

[8] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.