Department of Mathematics and Computer Science

University of Southern Denmark

A General Purpose Local Search Solver

Author:
Bo Stentebjerg-Hansen

Supervisor:
Marco Chiarandini

February, 2016

Acknowledgements

I have been very happy for studying at IMADA and would use this opportunity to thank all of the students here for creating this study environment. I would especially thank Marco Chiarandini for being my supervisor in this thesis. He is always found time to discuss interesting ideas and has provide many suggestions for this thesis. I would also like to thank my office "Domænet" for supporting me in tough times.
I am especially grateful for the support Rojin Kianian, Anders Knudsen, and Jonas Camillus has provide. Least but certainly not last I am very happy for all the support Lotte Rasmussen has given.

Contents

1 Introduction

The field of optimization can be split into several subfields depending on the nature of the decision variables (continuous or discrete) and the structure of the problem (linear, non linear, combinatorial, convex or 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 exact, 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 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 that allows use of local search, but the project is now abandoned.
OscaR is another CP based solver that uses local search to find a good solution. In this thesis, a general heuristic solver based on local search has been developed. It uses Gecode to find an initial solution and uses local search to try and 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.

Beside the basic components of local search several methods have been studied to see their effect on the solution. One of these elements is preprocessing, assisted by from Gecode, 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 efficiently represent 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 MIPLIB2010 and compared to Gurobi.

2 Introduction to 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. A dependent variable is only allowed to change value other variables it is dependent on change value.

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. An integer linear program (ILP) can be written 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 $\mathbf{c} \in \mathbb{R}^n$. (1) is the objective function and can easily be transformed to a maximization problem by multiplying by -1. The relation in (2) can be a mix of $\{\leq, =, \geq\}$, but greater than or equal can be transformed to less than or equal by multiplying both sides 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 points 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 formualtions are NP-hard [5, p.30] and several techniques have been developed for solving them. The techniques can be i.e. branch and bound, cutting plane and branch and cut [5, 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 [5, 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 uses either a programming language or a framework that have procedures implemented to solve the problem according to the constraints posted. The language or framework may provide global constraints that can be used to formulate the problem. An example of a global constraint is the alldifferent(**x**) constraint that specifies the variables $\mathbf{x}$ 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 [2, 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 [2, 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.

3.2.1 Gecode

Gecode (GEneric COnstraint Development Environment) is a constraint programming solver implemented in C++ and offers a wide range of modeling features. Gecode offers more than 70 constraints from the "Global Constraint Catalog" [7] that can be applied to boolean, integer, set and float variables.
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 change domain the corresponding view tells its subscriber 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 $O(D(x)^n)$. [9, p.57].
To solve a problem Gecode needs guidance when searching and that is done by a branching 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 variables to branch on first and which 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 [9, 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 SPACEs while searching for a solution that are used for backtracking. When Gecode reaches a failed SPACE, instead of starting from scratch and recompute all the 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 guarantee for performance.
To create a model for local search a set of variables and a set of constraints for the variables needs to be defined. The constraints are either implicit or soft. The variables and implicit constraints define the candidate solution and hence the search space S.

3.3.1 Implicit Constraints

Implicit constraints 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, for not violated and 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 $\mathbf{X}$ 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* $f(\tau)$ evaluates the quality of the candidate solution. The *neighborhood function* $N(\tau)$ that defines the candidate solutions s in the search space 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 reached 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.
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 variables 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. [6, p. 3-5]
Local search needs a *termination criteria* that determines when the search should stop. Sometimes we know what the optimal solution is, i.e. the SAT problem, if we find a feasible solution we can stop. In other cases an optimal solution may not be known, 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 commits 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 examines 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 a worse solution and/or infeasible solution and it 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, which 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 solutions 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 could be greedy heuristics like first fit or best fit.
Construction heuristics can be tailored 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 to get a better initial solution.

3.3.6 Metaheuristics

Metaheuristics defines how the search space should be explored, whereas local search algorithms focus more on the neighborhood of the solution. Iterative improvement often find a local optimum quickly, 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 [6, p.135]. Metaheuristics are used to get out of local optima to search different parts of the search space.
A simple metaheuristic 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 metaheuristic 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 a better 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 solutions in the tabu list are not considered 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 operations. 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 an *aspiration criteria* can be implemented. Aspiration criteria is a set of rules that can overrule the tabu list [6, p.139-140]. A common aspiration criteria is to ignore the tabu list 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 variables in each neighborhood

operation, which gives a very large neighborhood, but might need fewer iterations to find a good quality solution.

3.3.7 Creating a Model for Local Search

When creating a model for local search there are several things to consider, such as what the variables should represent and which constraints 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 solutions 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 increase efficintcy of local search, 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 solutions 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 contributes to the evaluation function. I.e. If a constraint has weight ten and a violation degree of three, it contributes 30 to the evaluation function. In this way the search prioritizes to satisfy certain constraints more than others. This method can have the negative effect that it might prioritize to have all constraint with violation degree close to zero instead of one constraint with a very high violation degree. If many constraints are violated it might take more iterations to get to a feasible solution than if only a single constraint is infeasible.
Another way is 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 introduces 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 being stuck 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 7.

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 the 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 objectives 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, which means that all constraint have some methods in common. This is especially useful when combining multiple constraints in a constraint system. 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 exists an objective interface.

The next layer is where the user models their problem and use the objects mentioned above. Several search procedures are implemented. The benefit of this architecture is that 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

Oscar is a another CBLS framework implemented in Scala and has drawn inspiration from Comet. It is implemented as a backend solver for MiniZinc, MiniZinc is a modeling language to combinatorial problems.
OscarR uses invariants and oneway constraints like Comet and uses propagation graphs. OscaR allow the user to define variables through oneway constraints and creates more oneway constraints defining variables if possible. The propagation graphs are split in a dynamic and static graph. The graphs have the same set of vertices but the dynamic has a subset of the edges of the static. In the dynamic graph cycles are not allowedbut they are in the static. If a cycles would be formed in the dynamic it is broken. In order to only update each invariant once the cycles in the static graph is consider a single vertex of a strongly connected component.
Their search procedure uses different neighborhoods for independent varibles and implicit constraints. Each variable in a model can only be involved in one neighborhood. Some of the neighborhoods are constraints specific meaning they are only uses if a specific constraint is used when modeling.
More information about OscaR can eb found here [3] [8].

4 Architectural Overview

This section gives an overview of the key components of this solver. Figure 1 gives an overview of the most basic classes and the classes they store pointers to i.e. have access to. The two engines of the framework are the GecodeEngine and LocalSearchEngine which 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 on 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 perfomed is described in section 7. LocalSearchEngine transforms the model to a model better suited for local search before the local search is started. 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 constraints and invariants respectively. They are described in subsection 4.2.1 and 4.3.2. They contain abstract methods which the subclasses must define.
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 creation of variables and constraints, finding initial solution and optimizing the solution and is described in subsection 4.4.

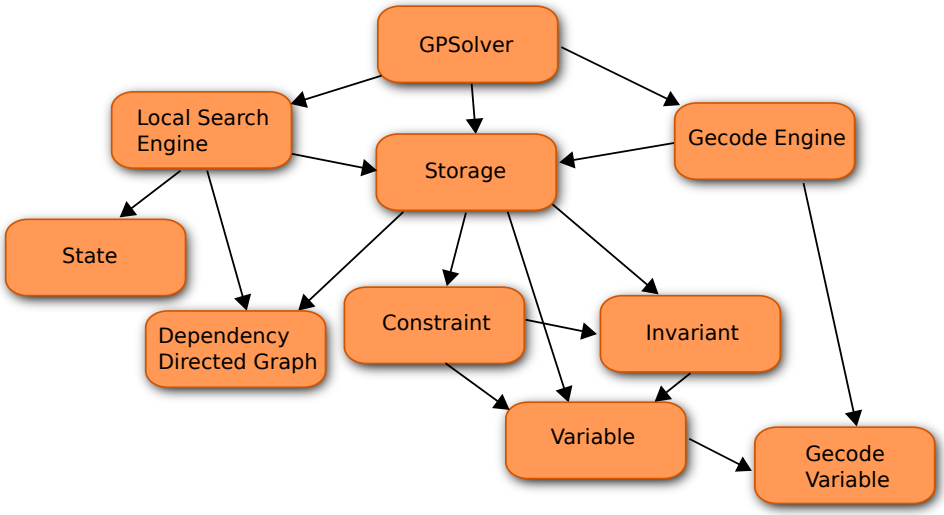

Figure 1: Overview of what the key classes of the framework have access to.

4.1 Variables

The VARIABLE only consist of short setter and getter methods. Each variable has a unique increasing id in the ordered by the time of creation. Their current value must be within their domain, given by a lower and upper bound. Each variable has a flag that tell if the domain size of the variable is equal to one and a flag that tell if it is defined by a oneway constraint. They have a counter 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) which 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 differentiate between infeasible solutions. 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 (5) is an example of a LINEAR constraint.

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

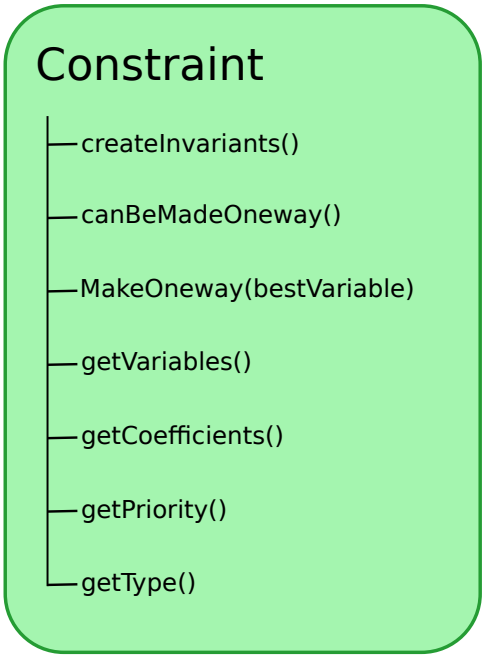

Figure 2: Important methods in Constraint class.

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. 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 whether the constraint is violated and the degree of violation. The degree of violation can be one if violated and zero otherwise, or it gives a measure of how violated the constraint is. For the linear constraint c_1 in equation (5) 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 the Linear Constraint

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 any of the reamaining is used it is transformed into a less than or equal relation instead. The change can be done by multiplying with minus 1 on each side to change from a "greater" to a "less" relation. All coefficients and variables are integers hence all strictly less/greater relations can be transformed to a less/greater and equal relation by increasing or decreasing $b(c_j)$ by one.
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.
If a variable is 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 implement 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 have a delta value, a current value, coefficient map, a stack of changes, a lower bound, and an upper bound. These are

used by the different methods of the invariants.
All Invariants must implement the methods proposeChange, calculateDelta , and updateValue which 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 used 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 use 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 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.

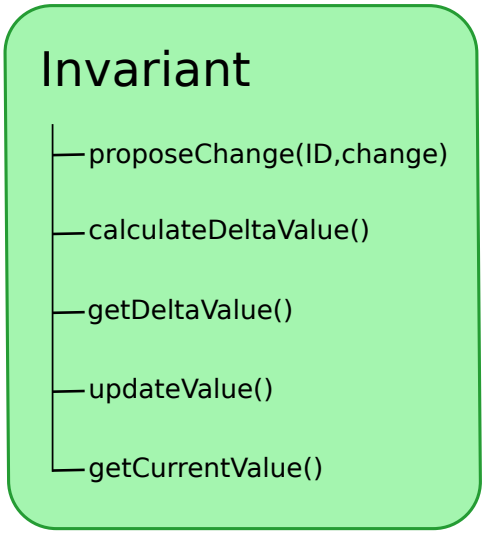

Figure 3: Important methods in Invariant class.

$x_1 = 2x_2 + 4x_3 + 1$, SUM can represent the right hand side in the equation defining x_1.
The method proposeChange(int *variableID*, int *valueChange*) uses the variables id to get its coefficient from a hashmap. The coefficient multiplied by the integer *valueChange*, the change of the variables value, gives the delta value that is pushed on a stack *variableChange*.
The method calculateDelta () first resets the delta value to zero and then pop 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 7.
It updates its value 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 relations 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 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 an instance that is described in subsection 5.2.
All constraints 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.
To implements a new constraint object it must be a subclass of 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 no initial solution is found within the limits given.

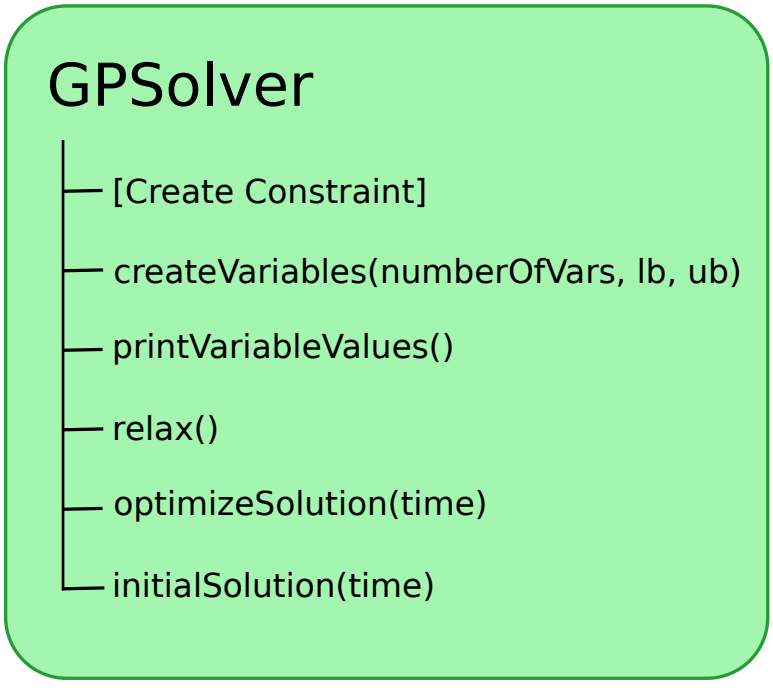

Figure 4: Important methods in GPSolver class.

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. This method starts the local search which is described in section 7.

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 [9, 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 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 completed a Gecode DFS search engine (Depth First Search) 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 an 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 objects, a MULTISTOP object has been implemented that combines all three stop objects such that it can have multiple stopping criterion.
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 GECODEENGINE is created and all variables and constraints, except those relaxed, are created and added to the new space. In order to choose which constraint 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 are. The reason for this choice is these constraints are used to create oneway constraints and are only created if the functional constraints are feasible.
The constraints are chosen by their priority and ties are broken at random. I.e. a model with 100 non-functional constraints of priority 3, 40 of priority 2, and 15 of priority 1 and consider the case where 20 constraints should be relaxed. The 15 constraints with priority 1 and 5 of the 40 constraints with priority 2, 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.
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 with Gecode has failed.
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 optimization problem (COP) has been found, the model is transformed to create a model suited for local search. Two new datastructures 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 handle dependencies of invariants when a variables value is changed. 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. This is done 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/or invariants.
The only constraint implemented, LINEAR, is used as an example for creating oneway constraints.

6.1 Simplification

The idea is to reduce the number of independent variables hence reducing the size of search space and 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 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 were allowed it could create non integer coefficients that 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()

```
    input : Constraint c_j
 1  Variable bestVariable = NULL
 2  int numberOfTies = 0
 3  foreach x_i in X(c_j) do
 4  |  // Break ties
 5  |  if defines(x_i) < defines(bestVariable) then
 6  |  |  // Choose the variable that helps define fewest
    |  |     invariants
 7  |  |  bestVariable = x_i
 8  |  |  numberOfTies = 0
 9  |  end
10  |  else if defines(x_i) == defines(bestVariable) then
11  |  |  if |deg(x_i)| < |deg(bestVariable)| then
12  |  |  |  // Choose the variable with lowest degree
13  |  |  |
14  |  |  |  bestVariable = x_i
15  |  |  |  numberOfTies = 0
16  |  |  end
17  |  |  else if |deg(x_i)| == |deg(bestVariable)| then
18  |  |  |  // Fair random
19  |  |  |  numberOfTies++
20  |  |  |  if Random(0,numberOfTies) == 0 then
21  |  |  |  |  bestVariable = x_i
22  |  |  |  end
23  |  |  end
24  |  end
25  end
26  if  bestVariable ≠ NULL then
27  |  makeOneway(Variable bestVariable)
28  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 participates 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 the best variable is found, if any, algorithm 6 `makeOneway` is called.

Algorithm 6: Linear - makeOneway(Variable x_i)

input : Variable x_i, **Constraint** c_j
output: A new invariant y defined by a oneway constraint

```
1  set Q = ∅                                   // new coefficient set
2  set U = ∅                                      // new variable set
3  // Move other variables to right hand side and update
       their cofficient
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  // Isolate x_i
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 y and removing the constraint c_j and variable x_i.
The value of the invariant must always be within its domain, corresponding to domain of the variable. It is not allowed to change value of one of the variable such that the invariants value is not within its domain.

6.2 Dependency Digraph

For each constraint, not transformed into a oneway constraint, auxiliary variables are introduced as invariants and an invariant for violation degree is created. They are created by calling the method createInvariants of each constraint and the algorithm for LINEAR was shown in subsection 4.2.1.
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 $vu \in A$ 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.

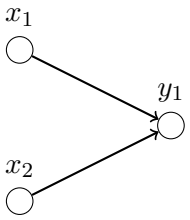

Figure 5: Small example of DDG.

The DDG is used when doing local search and contains dependencies of variables and invariants. 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 [10, p. 97] and OscaR[3, 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 is 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. This is illustrated in figure 5.

Auxiliary variables can be useful to update constraint violations and in this example we could create an auxiliary variable where the 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. A LEQVIOLATION invariant y_3 representing the violation of constraint c_2 is added as well. 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.

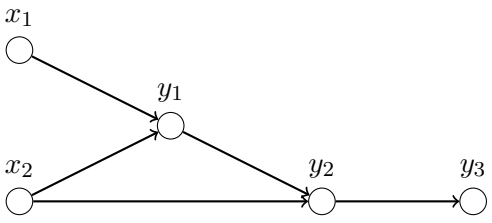

Figure 6: Small example of DDG continued.

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 which 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 $(u, v) \in V^{SCC}$ there exist both a path from u to v and a path from v to u [4, p. 1170]. Each of these strongly connected components must be broken in order to keep G acyclic, since an SCC consists of at least one cycle. An 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 set of vertices V^{SCC}, in an SCC, can only be vertices representing invariants since variables 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 the oneway constraint defining it. If there are ties, they are broken at random.
To find all SCC's of size two or more, Tarjan's algorithm for finding strongly connected components is used. . Tarjan's algorithm has a recursive 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. It uses two counters; an *index* which is unique in the order they are visited, and *lowlink* is initially the same as *index*. If we go from vertex v to vertex u and u has been visited before, $v.lowlink = min(v.index, u.lowlink$. If u has not been visited the search continues from u. When all neighbors of a vertex v has been explored if $v.index$ is equal to its $v.lowlink$ the stack is popped until v is popped. All the vertices V^{SCC} popped from the stack is a strongly connected component.
The algorithm has been modified to give each variable a time stamp when they are popped from the stack as well. 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 restarted. Removing the fewest possible invariants such that there are no cycles in G

corresponds to solving the *minimum feedback arc set problem* and is known to be NP-hard [3, 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. After they are added Tarjans algorithm is repeated to give all invariants a time stamp.
The first tiebreaker in algorithm 6 is used as a heuristic to reduce the number of cycles generated.

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. 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.
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 7 shows the necessity of such an ordering.

If y_1 is updated before y_2 then it might need to be updated again after y_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.

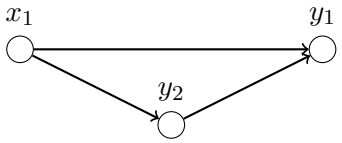

Figure 7: Importance of propagation queue

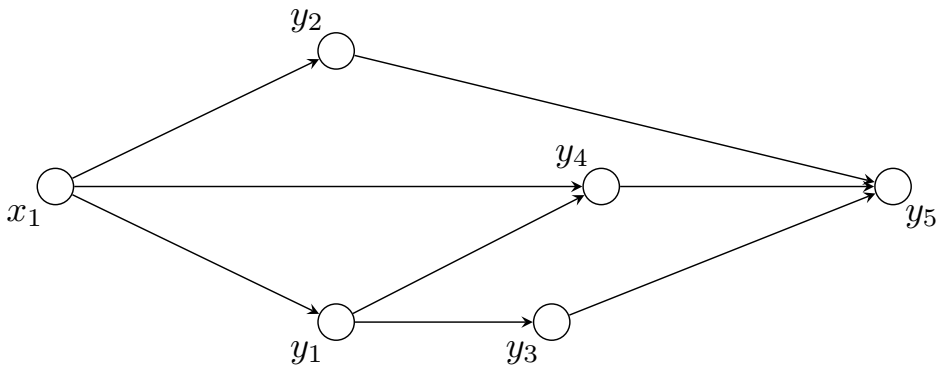

Figure 8: Example of subgraph in the DDG.

Once the dependency directed graph is a DAG each invariant vertex has 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 v_i in dependency digraph G a depth first search is made. Each vertex visited is added to the propagation queue of v_i. This gives a time complexity of $O(rlog(r))$ where r is the number of reachable vertices from v_i.

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 can be merged into a single queue removing duplicates.
Figure 8 and 9 gives an example of creating a propagation queue for a variable based on the dependency graph. It is possible to create different propagation queues from the example but we do not differentiate between them as long as there is no backward going arc. In this case we have single variable x_1 that creates the propagation queue $q_1 = [y_2, y_1, y_4, y_3, y_5]$. The idea is to send the changes from x_1 to y_2,y_1, and y_4 then go to y_2 update its value and send the change to y_5. Go to y_1, update and send change, go to y_4 etc.

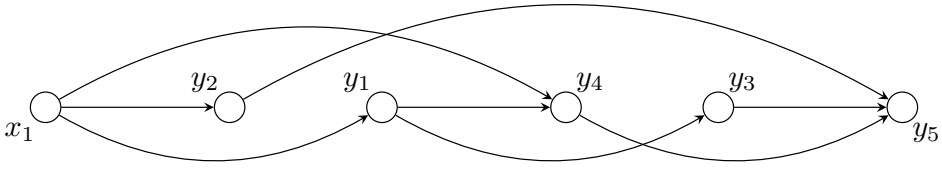

Figure 9: Example of a propagation queue created from figure 8.

7 Local Search and Metaheuristics

The LocalSearchEngine class uses the model created for local search described in section 6 and uses local search to improve the initial solution. The vector holding pointers to the independent variables is given a random ordering, called a mask, that is used in local search.
Local search explores how changing the value of few variables will affect the solution quality, hence exploring a neighbor solution. The key for being efficient is to compute this change fast. The dependency digraph and the propagation queues are used for this. Before committing a neighborhood operation several neighborhood operations might be explored before choosing one to commit. 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 committed. By this we can evaluate the neighbor solution without committing a neighborhood operation.
Each constraint $c \in C$ created in the model was given a priority p and these priorities are used during local search. Let k denote the highest priority given. A new Sum invariant is created for each priority p and one for the objective function. This is done at the same time invariants are created by each constraint class.
Let $q_p \in Q$ be the sum of violation degree of the constraints with priority p. The objective functions value is consider as q_0 and the vector Q is used to evaluate the quality of a solution. The evaluation function $f(\tau)$ is not returning a single value but the quality vector, $f(\tau) = Q_\tau$.
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 time they differ in 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 constraints with priority 2 is consider. Then candidate solution τ' is consider better than τ since $Q_\tau(2) = 4 > Q_{\tau'}(2) = 3$.
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. A subclass of the Neighborhood class is the choice of neighborhood function and gives the sequence in which the neighbor solutions in the neighborhood graph are explored. The search procedures can query a Neighborhood class to evaluate a neighborhood operations, a Move class, effect on the

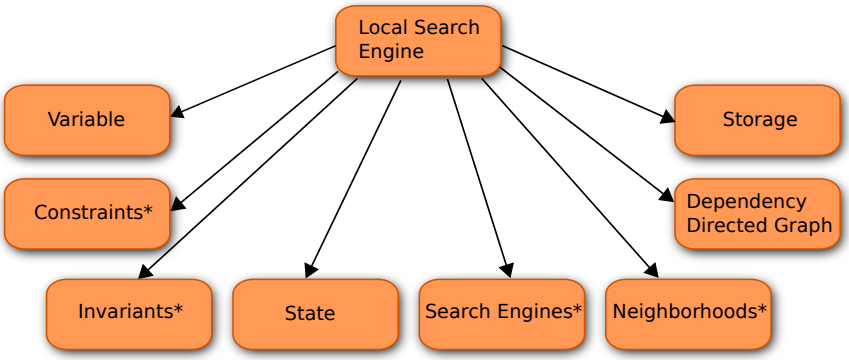

Figure 10: Overview of the class pointers of LocalSearchEngine. The fields marked with a star (*) are several classes of that type.

evaluation function. The search procedures determine which neighborhood operation to commit, if any. Neighborhoods and search procedures are combined to create different local search algorithms and LocalSearchEngine uses them within the time limit to search for a better solution.
The first algorithms that is used is a first improvement until a local optmia is found. Gecode is not used to optmize hence it might be posible to improve the initial solution very easily. If Gecode does not find an initial solution it is very likely that the construction heuristic, random assignment, does not give a feasible solution and is far from a local optima. It is prefered to start in local optima or at least close to one and first improvement is very efficient at finding a local optima.

7.1 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 11.

All Neighborhoods implemented use a step function 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 object 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 1.

The method `next()` creates new Move object and returns a pointer to it. If all possible neighborhood operation of the neighborhood function have been returned the method returns a pointer to **NULL** instead. To know when a neighborhood has been fully explored, counters and iterators are used de-

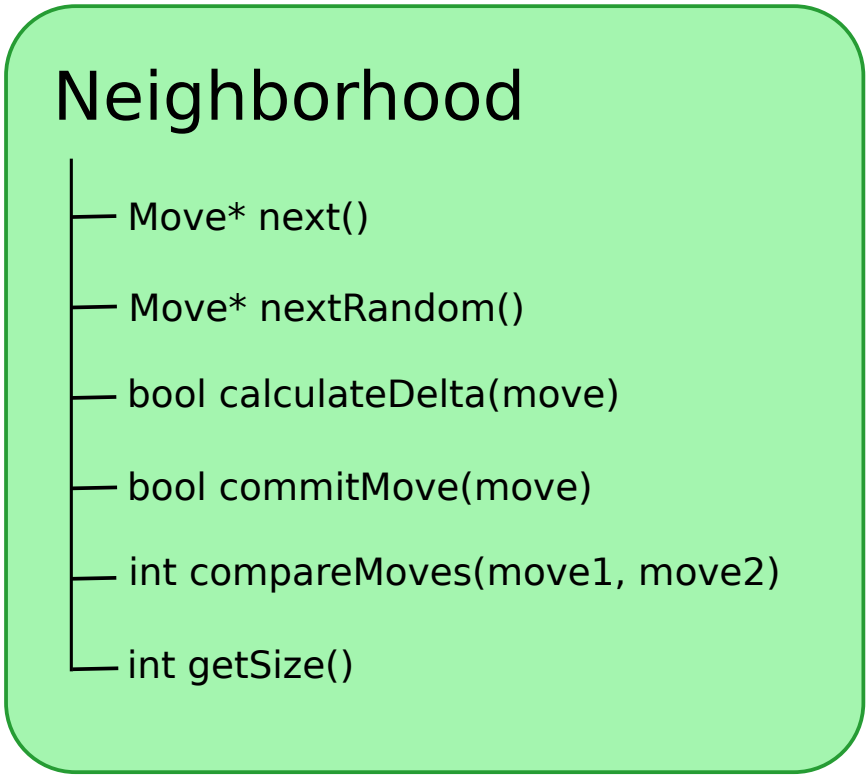

Figure 11: The methods all Neighborhood classes needs to implement.

pending 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.
For FLIPNEIGHBORHOOD when next() returns a MOVE and the variable chosen is given by a random sequence, a mask. When next() is called in the RESTRICTEDFLIPNE class it returns the next variable probability $p = \frac{5000}{n}$, where n is the number of independent variables. This gives expected 5000 variable before it return a **NULL** pointer.
For CONFLICTONLYNE the neighborhood operations consist of a variable

class	Heuristic
FLIPNEIGHBORHOOD	All variables
RESTRICTEDFLIPNE	An expected 5000 variables chosen at random
CONFLICTONLYNE	All variables in unsatisfied constraints
RANDOMCONFLICTFLIPNE	Variables from a random unsatisfied constraint

Table 1: Table of NEIGHBORHOOD classes.

that participate in a constraint that is unsatisfied.
The class RandomConflictFlipNE chooses an unsatisfied constraint at random and returns a Move with a variable participating in that constraint until each Move with a different variable has been returned.
Method nextRandom() returns a Move with a random variable from the neighborhood.
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 is identical for all the Neighborhood classes implemented. The method returns a flag that indicates if the suggested neighborhood operation is forbid by a oneway constraint.
The following describes how calculating the delta change of a variable x_i changing value.

1. Reset delta value of invariants in quality vector Q
2. Send delta value of x_i to neighbor invariants in DDG.
3. For each invariant y_j in propagation queue of x_i, calculate delta value y_j, if it is not zero, send the change to neighbors 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. Then return false.
5. Otherwise update delta quality vector of move.
6. return true if the *move* is an allowed neighborhood operation.

The delta values of the invariants can be reset by calling calculateDelta , when no change is send. The reason for resetting them is only to make sure their stack of changes are empty before the next neighborhood operation is calculated.
To commit a neighborhood operation the method commitMove(move) is called with a pointer to the Move *move* that is wanted. commitMove(move) recalculate the delta value by calculateDelta (move) since other neighborhood operation might have been explored since move was calculated last. Once the delta values of invariants have been computed, the invariants can be updated by calling their updateValue(). 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. If the current solution is feasible their neighborhood sizes are zero.
A default method compareMoves(move1,move2) compares the delta vector

of two MOVE classes and returns 0 if they are the same, 1 if move1 is best and 2 otherwise.
The size of the neighborhood with 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 iteration, for the others it is a constant size.

7.2 Search Procedures

NEIGHBORHOOD classes do not implement any strategy of which neighborhood operation to choose. 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 a tabu tenure, and an aspiration criteria.

The class BESTIMPROVEMENT looks at each MOVE a NEIGHBORHOOD class NE gives and finds uses the MOVE that leads to the best solution. The best MOVE is determined from their delta vector after the method calculateDelta () of the NEIGHBORHOOD is called on each MOVE returned by the neighborhood until **NULL** is returned. BESTIMPROVEMENT returns a flag that tells if the current solution was improved. A flag 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, NEIGHBORHOOD NE
output: bool improvement
1  MOVE bestMove = NE.next()
2  MOVE move = NE.next()
3  while move ≠ NULL do
4  |  bool allowed = NE.calculateDelta(move)
5  |  if !allowed then
6  |  |  move = NE.next()
7  |  |  continue
8  |  end
9  |  bestMove = compareMove(move,bestMove)
10 |  move = NE.next()
11 end
12 bool improvement = Check if bestMove gives improvement
13 // by looking at delta vector
14
15 if improvement OR alwaysCommit then
16 |  NE.commitMove(bestMove)
17 end
18 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 an implementation very similar to BESTIMPROVEMENT. 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 class.

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 benefit is many iteration can be done but they are not as 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 iterations made so far, 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. The aspiration criteria is if a neighborhood operation is tabu but leads to a solution better than one found so far, the tabu list is ignored and that neighborhood operation is done.

The tabu list has size n and each variable has a integer corresponding to the last iteration that variable was changed. The tabu tenure is chosen to be based on the neighborhood size with a small degree of random. $tt = random(0, 10) + min(2 \cdot neigborhoodsize, n/200)$. The neighborhood size might be very small when only considering variables that are in a conflicting constraint, hence the multiplier.
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
 1  int tabuTenure = Random(0,10)+ min(NE.getSize()*2,
    tabulist.size() /200)
 2  MOVE bestMove = NE.next()
 3  MOVE move = NE.next()
 4  while move != NULL do
 5      bool allowed = NE.calculateDelta(move)
 6      if !allowed then
 7          move = NE.next()
 8          continue
 9      end
10      bool isTabu = (iteration - tabulist[move.ID]) <= tabutenure
11      if isTabu then
12          if betterThanBest(current,move.getDeltaVector(), best)
            then
13              NE.commitMove(move) tabulist[move.ID] = iteration
14              return true
15          end
16          move = NE.next()
17          continue
18      end
19      bestMove = compareMove(move,bestMove)
20      move = NE.next()
21  end
22  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.

7.3 Local Search Algorithms

When a model better suited for local search has been made, the remaining time is used to do local search. The algorithms check if the timelimit is reach before each iteration of a search procedure. The best solution found while searching is saved in a STATE class such that the search can continue but always report the best solution when the time limit is reached.
Three algorithms have 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 iterations made in case the neighborhood is large.

Algorithm 9: Local Search - Test Algorithm 1

```
1  ConflictOnlyNE CON
2  RestrictedFlipNE RFN
3  TabuSearch TSCON(NE)
4  TabuSearch TSRFN(NE2)
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 iterated local search using first improvement and random walk [6, p. 141-142] with a single flip neighborhood. They

idea is to find a local optima fast, and use randomness to escape the optima. The random walk chooses a random neighborhood operation and commits that if it is legal. This is repeated a number of time depending on the size of the neighborhood. This is done to try to explore many different subspaces of the search space S.

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  while within time limit do
6  |  bool improvement = true
7  |  while improvement AND within time limit do
8  |  |  improvement = FI.start()
9  |  |  if getSolution() is better than bestSolution then
10 |  |  |  bestSolution = getSolution()
11 |  |  end
12 |  end
13 |  RW.start()
14 |  if getSolution() is better than bestSolution then
15 |  |  bestSolution = getSolution()
16 |  end
17 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 RCC
2  RestrictedFlipNE RFN
3  BestImprovement BIRCC(RCC)
4  TabuSearch TSRFN(RFN)
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.

8 Testing the Framework

There are many parameters that can be tested in this framework and only some of them has been selected. The test are shown in chronological order, meaning the results of one test is when creating the next test. To test the difference between results a Wilcoxon signed-rank test is used, which available in R. The test have been done on a subset of the instances from MIPLIB2010. The test is done on Horseshoe9 and the hardware specification can been seen here [1].

8.1 Parameters of Gecodes DFS Search Engine

The search engine was described in section 5 and the MULTISTOP object can be used to stop the search for a solution. The search can be stopped based on three parameters, time, number of fails, and number of nodes explored. The number of nodes explored is highly correlated to the instance size and will not be tested. All the instances have been tested with a time limit of 100 seconds and random seed 60. This is done to get an indication of the time Gecode uses and the number of failed spaces before Gecode finds a solution, if any.
The goals of the test are to find a parameter for time and number of fails, such that the search can be stopped early if it is likely it will not find a solution. The worst case is Gecode never finds a solution hence the time used for searching has not be useful.
The result of the time is shown in figure 12 on the next page. The dots are blue if a solution has been found and red if Gecode did not find a solution. In all the instances where a solution was found it was found within 10 seconds. Based on this the time Gecode search for a solution is reduced to 10 seconds. The number of failed spaces can be seen in figure 13 on the page after the time test.
In almost all of the instances in which a solution was found, there were zero spaces where failed. The only exception is “neos-1440225” that reported 19118 spaces to be failed. One of the instances where solution was not found reports that zero spaces were failed but otherwise they all report a high amount of failed spaces. Based on this test the number of failed spaces tolerated is set to 1. This will exclude one initial solution but save time on a lot of the others.
There are other parameters for Gecode that could have been tested is the different ways for Gecode to branch.

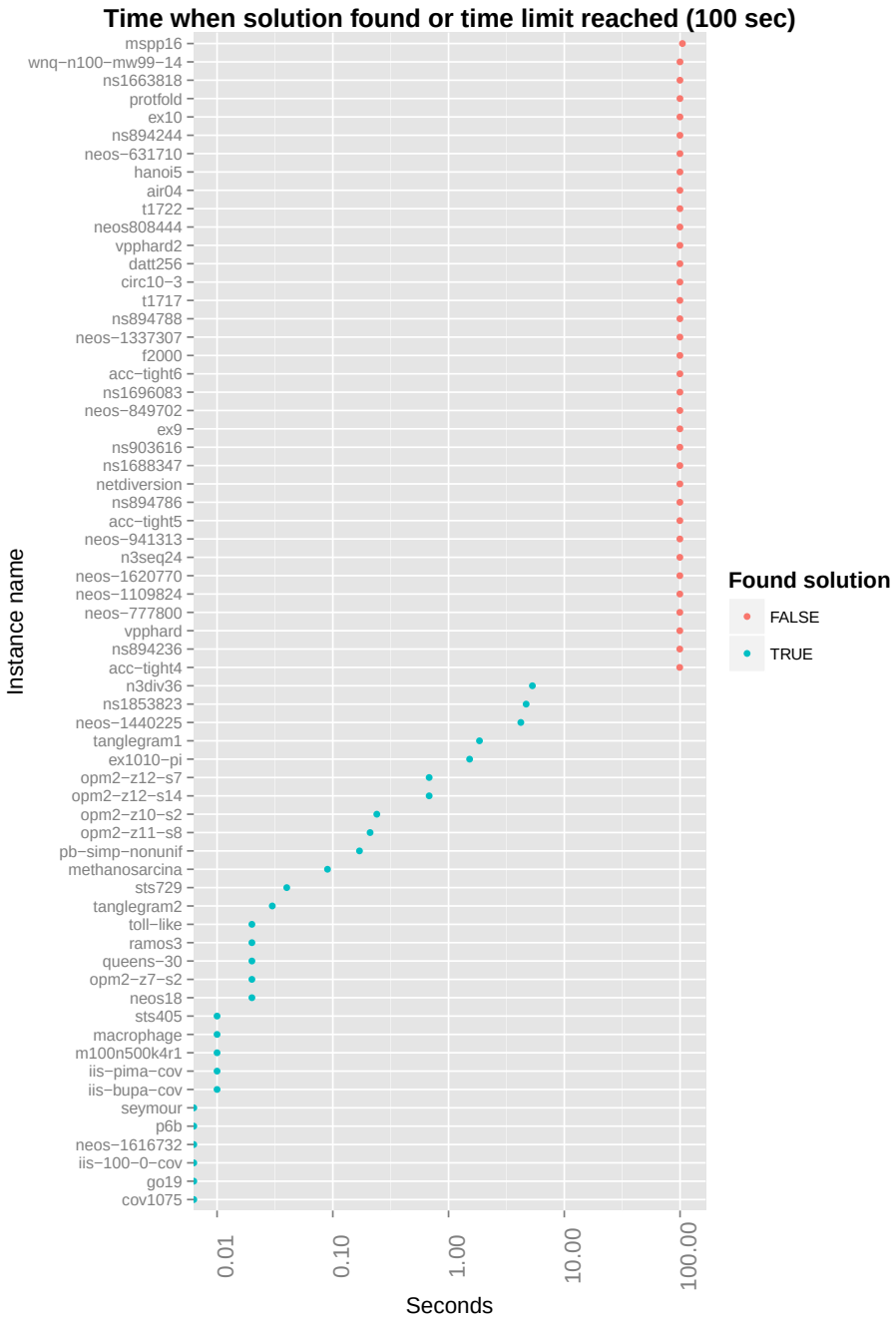

Figure 12: The seconds are on a logarithmic scale.

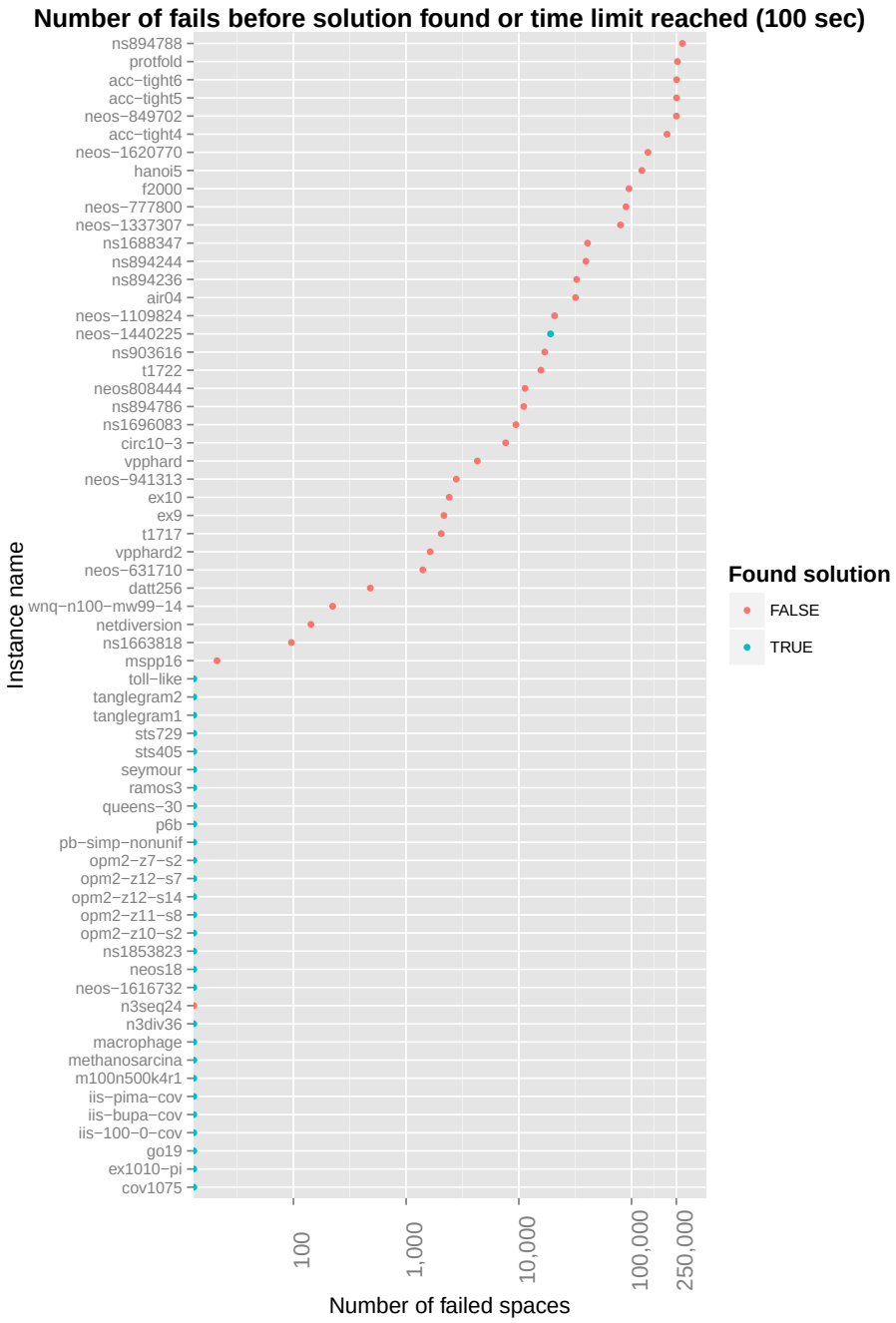

Figure 13: The number of failed spaces are on a logarithmic scale.

8.2 Defining variables by Oneway Constraints

If the model contains functional constraints we try to define one variable for each functional constraint to reduce the search space. The effect has been test on the instances where functional constraints are present. They have been tested by algorithm 1, Tabu search with ConflictOnlyNE and tabu search with FlipNeigborhood, with a total run time of 120 seconds. The seed that was used during the test was 42.
The resulting number of violations are shown in figure 14. To see if defining variables by oneway constraints have an impact on the result it has been analyzed with Wilcoxon signed-rank test. The p-value of the test is 0.8888 hence it is not statistical significant that one result was better than the other.
The two cases where the number of violations are zero for both, the objective function is zero as well.

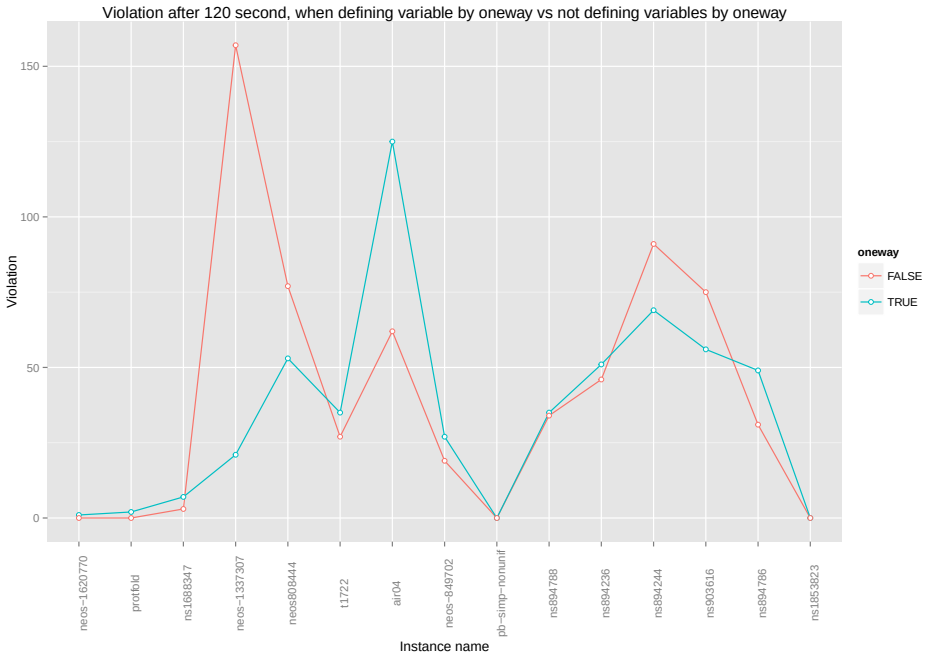

Figure 14: Sorted by number of defined variables in increasing order.

8.3 Using Gecode as Construction Heuristic

Gecode set limits on the design but it might provide a useful preprocessing and initial solution as well. We test the effect of using Gecode to find and initial solution versus a random assignment to the variables. In both cases a first improvement is used until local optima just like describe in section 7.

Two test on 46 instances has been made to test the effect of Gecode.
The first test algorithm 2, Iterated local search, is used until a total time limit of 30 seconds, with a random seed 42. The limit is chosen to make sure local search has started but not been running for very long. The violation of the result is shown in figure 15. There is barely a difference between the two

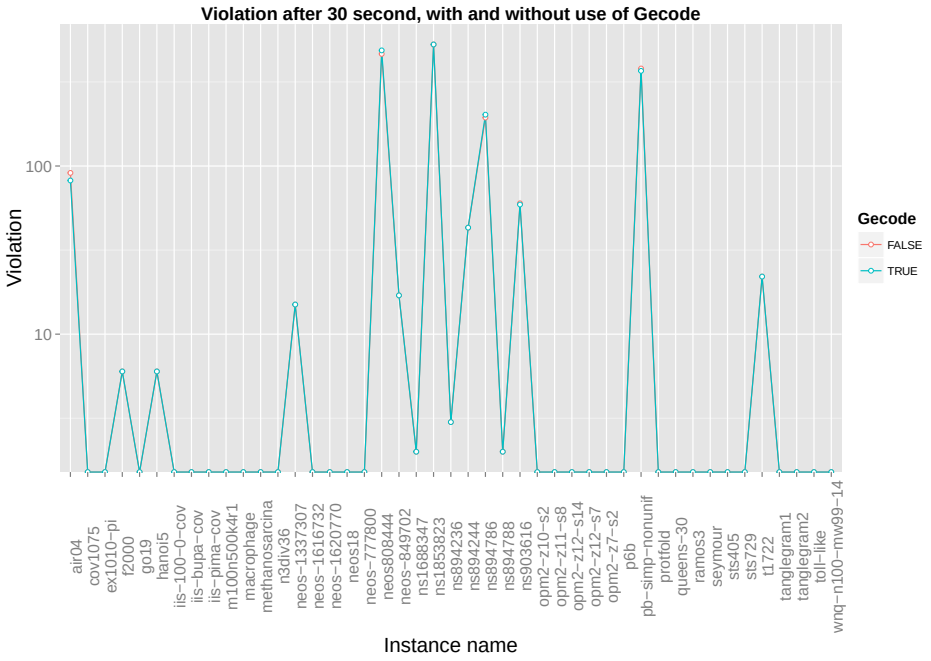

Figure 15: Violation after 30 seconds with algorithm 1, with and without Gecode

results and a Wilcoxon signed-rank test gives a p-value of 1, hence they are not significantly different. In order to look closer at the difference we look at the objective to see if there is a difference there. In order to visualize the data a ratio is created for each result, since the objective value span from -206179 to 3810000 in different instances.

$$ratio = \frac{obj.val1}{obj.val1 + obj.val2} \tag{8}$$

The objective value $obj.val1$ is the result without Gecode and $obj.val2$ is the result with gecode. This can only be done if the denominator is not zero and the values do not have opposite signs. In only two instances this was the case and they have been left out in the visualization. The two instances are "neos808444" and "neos-849702" and both have objective value zero. If the resulting ratio is 0.5 then the objective value is the same in the test. The result is shown in figure 16 and a Wilcoxon test gives a p-value of 0.8379. The figure show they give an equal objective value most of the time. This

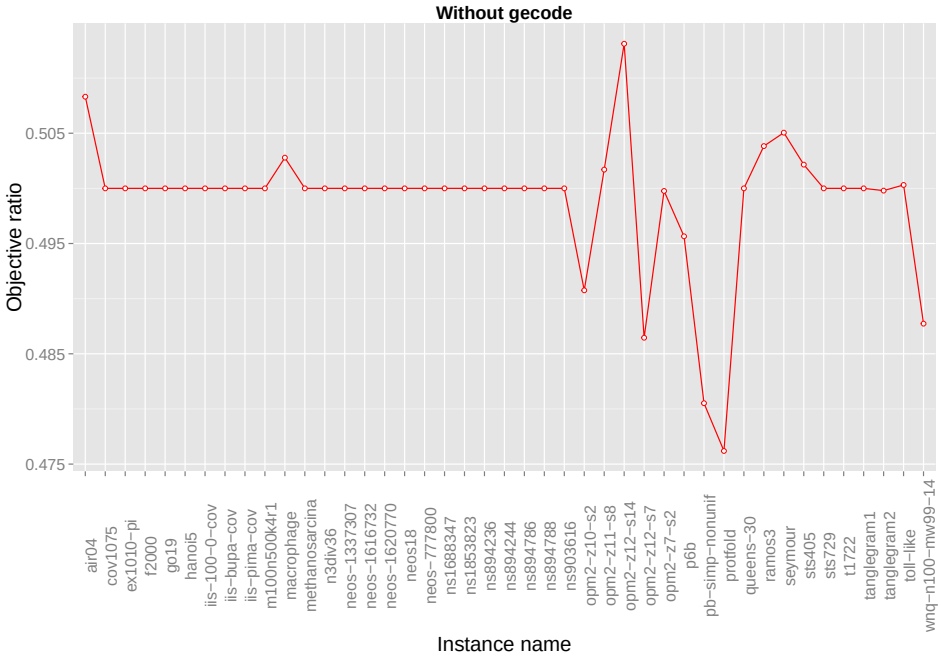

Figure 16: Ratio of objective value after 30 seconds with algorithm 1.

could be because the local search evened out the difference, hence the next test we leave out the local search.

The test will be Gecode and first improvement versus random assignment and first improvement, in both cases until a local optima has been found. In this case the run with Gecode proves to be significant better than without Gecode. Wilcoxon test gives a p-value of 0.01894. The result of the test is shown in figure 17.
Based on the last test we keep using Gecode for further test, though the time usage is of interest as well. The time use is plotted in figure 18 and it shows Gecode sometimes uses more time. The Wilcoxon test gives a p-value of 0.005979 so we can conclude the time use is significantly different. The red line indicate which of the runs that uses the most time.

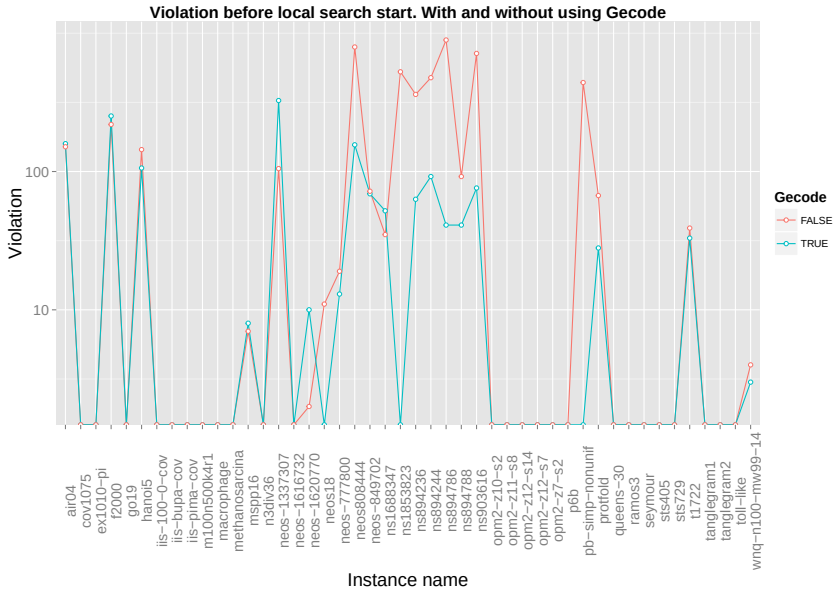

Figure 17: Violation after initial solution and first improvement, with and without Gecode.

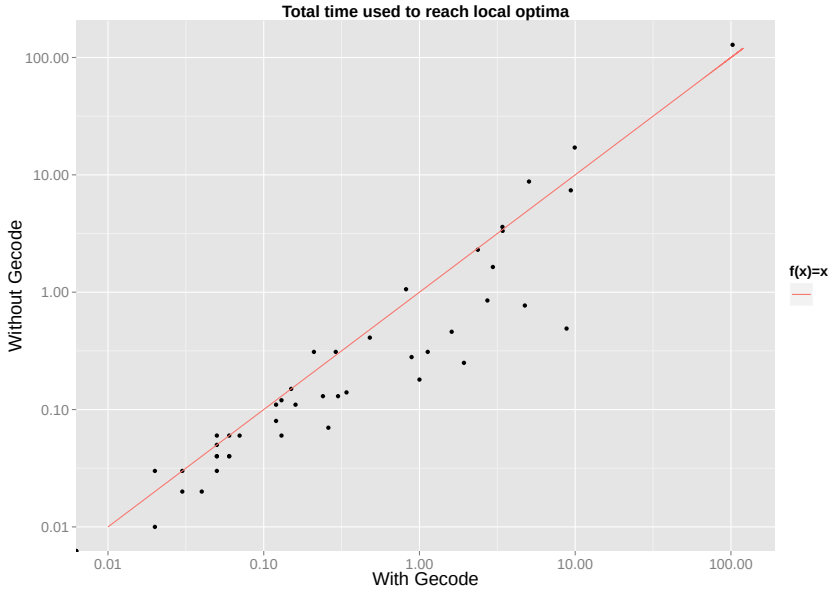

Figure 18: Time used before starting local search algorithms.

8.4 Testing the Algorithms

Algorithm one is using tabu search only considering independent variables that are in a conflicting constraint as long the the current solution is infeasible. When the solution is feasible if there is less than 5000 it considers all independent variables, otherwise it selects expected 5000 at random before choosing the best.
Algorithm two is an iterated local search, first improvement and random walk. The third algorithm uses a minimum conflict heuristic as long as the current solution is infeasible. When feasible it uses the same tabu search as algorithm one.
They have been tested on all 64 instances and will primarily be compared based on the number of violations. The test time used is 120 seconds and the random seed is 42. Based on the previous test Gecode is used with the parameters found in subsection 8.1. We try to define as many variables as possible as oneway constraints.
To illustrate the data one instance has been left out to make the other data more visible. Instead the data for that instance, "netdiversion", is shown in table 2. The data show that algorithm three is by far the best one. The minimum conflict heuristic is very effective but after 77 seconds it does not improve the solution anymore.
The result for the other instances is illustrated in figure 19 with a logarithmic scale on the number of violations. Even though algorithm three performed much better than the other on "netdiversion" it does perform much worse than the others on "neos-1337307". From the graph it looks like algorithm two performs a bit worse than the others. The algorithms has not been fine tuned but the parameters are based on a few initial test and what is found in other literature. The parameters have significant impact on the performance of an algorithm.

	objective	violation	iterations
Algorithm 1	5713042	3112	685
Algorithm 2	5712891	3470	1014
Algorithm 3	5714768	20	9773710

Table 2: Result on instance "netdiversion".

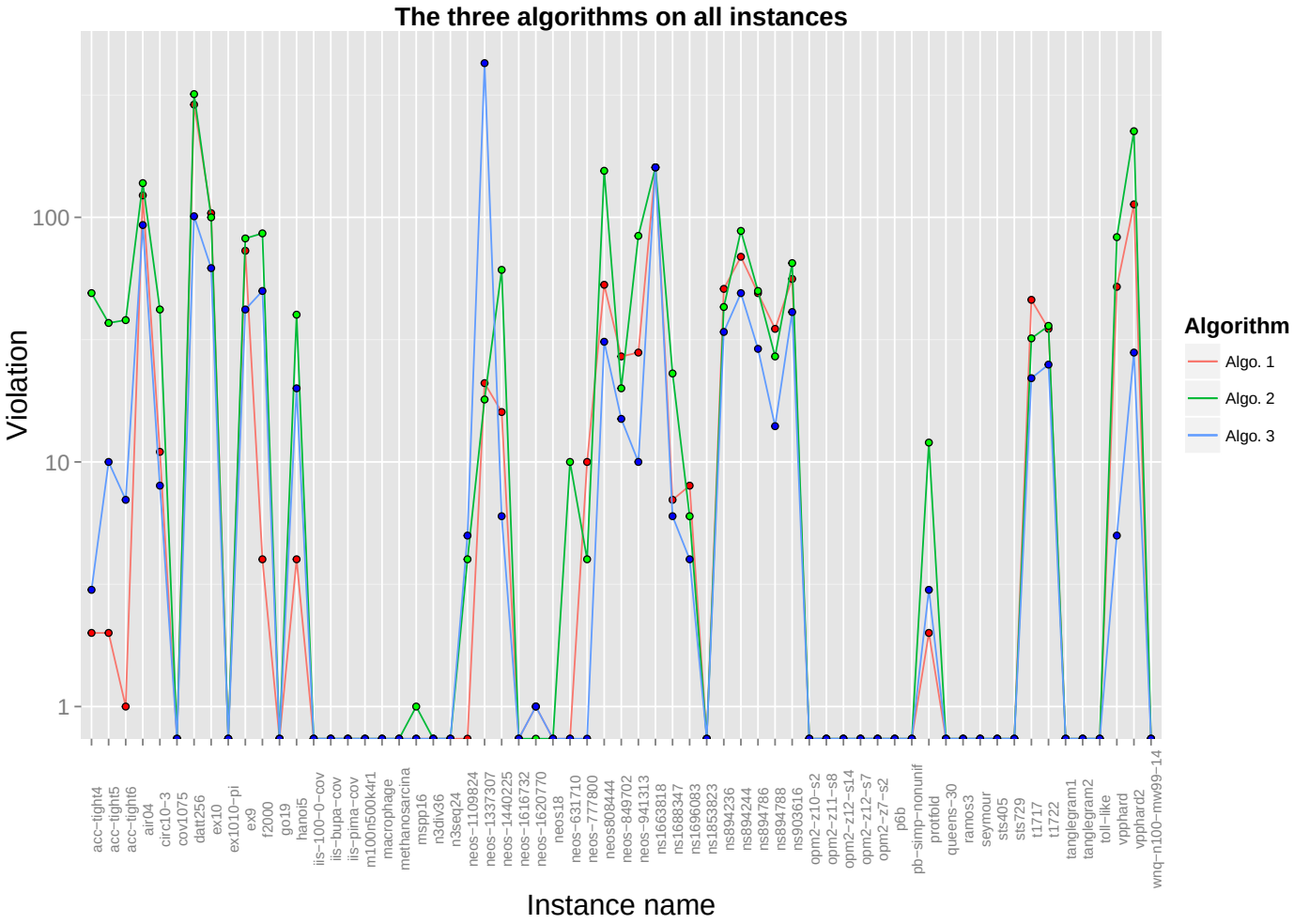

Figure 19: Violation after 120 seconds with the different algorithms.

The algorithms have been ranked on each of the 64 instances in order to find the best algorithm overall. The result is shown in figure 20 and the ranks are shown in box plots. The value they have been ranked by is $1.000.000 \cdot \textit{violations} + \textit{objective value}$ to make violation more important than the objective value. The algorithms have been given the rank 1,2, or 3 for each instance. In case of a tie they are given the average. i.e. If two algorithms are tied for second place they are given rank 2.5 each.
From the plot it looks like algorithm three is the best performing wining in at least 25 % of the instances. Algorithm two did perform worst in at least 50 % of the instances.

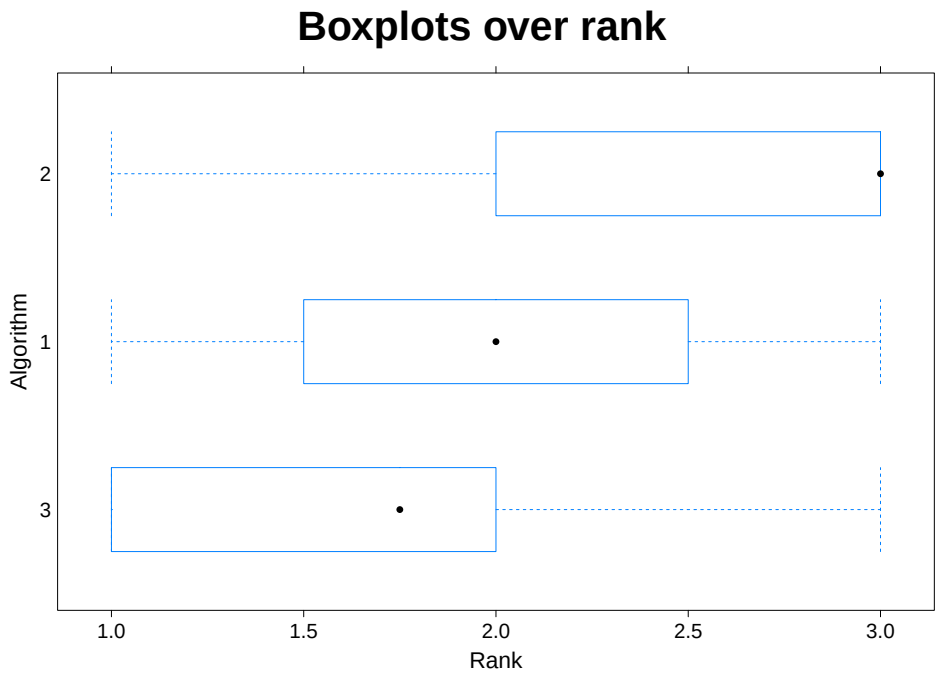

Figure 20: Boxplots over the ranking of the algorithms.

8.5 Versus Gurobi

The time limit of the tests are set to 120 seconds with random seed 42 and algorithm three is used. Gurobi has the same time limit and uses only one thread, as this framework do.
The tables on the next three pages show the final result when comparing to Gurobi. In the columns are instance name, number of variables, number of constraints, how many percentage Gecode solved for initial solution, and the time for finding initial solution. The next columns are the time used for finding a feasible solution, the value of the feasible solution, number of iterations with local search. The last four columns are final objective value, final number of violation, Gurobis solution value and the optimal value. If a feasible solution is not found the cells are given NA. If a optimal solution is not know, then the cell will contain NA.
Gurobi found a feasible solution in 52 instances, algorithm three found a feasible solution in 33 instances. In three instances algorithm three found a solution equal to Gurobi and in three instances it found a solution better than Gurobi.
Gurobi do better preprocessing than Gecode but Gurobi has been optimized towards MIPLIB instances. Based on the result the framework can match Gurobi on a few instances though Gurobi is far superior.
The source code is available at `https://github.com/Boste/Speciale.git`

Instance	#Vars	#Cons	Gecode Sol.	Init. Sol. Time	Feas.Time	Feas.Val	#Iter	Final Obj	Final Viol	Gurobi	Opt
acc-tight4	1620	3285	0	0.010000	NA	NA	7883990	0	3	0	0
acc-tight5	1339	3052	0	0.010000	NA	NA	7986211	0	10	0	0
acc-tight6	1335	3047	0	0.010000	NA	NA	8008028	0	7	0	0
air04	8904	823	25	0.800000	NA	NA	207078	76428	120	56137	56137
circ10-3	2700	42620	0	0.410000	NA	NA	243144	470	8	NA	NA
cov1075	120	637	100	0.010000	0.010000	42	99555	21	0	20	20
datt256	262144	11077	0	24.550000	NA	NA	153905	258	101	NA	NA
ex1010-pi	25200	1468	100	1.720000	1.720000	742	8189	449	0	244	NA
ex10	17680	69608	0	5.440000	NA	NA	39155	99	63	100	100
ex9	10404	40962	0	1.870000	NA	NA	75425	81	42	81	81
f2000	4000	10500	50	0.110000	NA	NA	10475255	1971	50	NA	NA
go19	441	441	100	0.000000	0.000000	147	527525	86	0	84	84
hanoi5	3862	16399	50	0.120000	NA	NA	11871831	1934	20	NA	1931
iis-100-0-cov	100	3831	100	0.020000	0.020000	59	33278	29	0	29	29
iis-bupa-cov	345	4803	100	0.030000	0.030000	129	16437	38	0	36	36
iis-pima-cov	768	7201	100	0.070000	0.070000	214	6806	35	0	33	33
m100n500k4r1	500	100	100	0.000000	0.000000	0	449501	-23	0	-24	-25
methanosarcina	7930	14604	100	0.150000	0.150000	5046	9835	5046	0	2916	NA
macrophage	2260	3164	100	0.030000	0.030000	609	35803	508	0	374	374
mspp16	29280	561657	50	51.390000	145.52	451	0	451	0	418	363
n3div36	22120	4484	100	4.540000	4.540000	480400	6160	480400	0	132800	130800
n3seq24	119856	6044	0	34.360000	41.04	841200	1465	841200	0	52200	52200

Table 3: Table of first 22 instances.

Instance	#Vars	#Cons	Gecode Sol.	Init. Sol. Time	Feas.Time	Feas.Val	#Iter	Final Obj	Final Viol	Gurobi	Opt
neos-1109824	1520	28979	0	0.180000	NA	NA	70641	808	6	378	378
neos-1337307	2840	5687	50	0.070000	NA	NA	694423	-201273	427	18	-202319
neos-1440225	1285	330	0	0.020000	NA	NA	3484127	36	6	36	36
neos-1616732	200	1999	100	0.000000	0.000000	176	277059	159	0	161	159
neos-1620770	792	9296	50	0.030000	NA	NA	11363096	18	1	9	9
neos18	3312	11402	100	0.040000	0.040000	22	10196	22	0	16	16
neos-631710	167056	169576	0	21.280000	24.03	555	9978	555	0	203	203
neos-777800	6400	479	50	0.230000	4.05	-80	14719	-80	0	-80	-80
neos808444	19846	18329	50	1.490000	NA	NA	3933914	0	43	0	0
neos-849702	1737	1041	25	0.030000	NA	NA	987431	0	17	NA	0
neos-941313	167910	13189	0	29.270000	NA	NA	5668704	19010	10	9361	9361
netdiversion	129180	119589	0	23.980000	NA	NA	9773710	5714768	20	400	242
ns1663818	124626	172017	0	44.190000	NA	NA	0	0	165	NA	86
ns1688347	2685	4191	25	0.100000	NA	NA	111441	35	6	270	27
ns1696083	7982	11063	0	0.610000	NA	NA	61680	53	4	NA	45
ns1853823	213440	224526	100	3.270000	3.270000	504000	4041	504000	0	467000	NA
ns894236	9666	8218	50	0.090000	NA	NA	1415963	18	35	NA	NA
ns894244	21856	12129	25	0.860000	NA	NA	983731	16	50	NA	15
ns894786	27278	16794	50	0.380000	NA	NA	1390731	14	29	NA	NA
ns894788	3463	2279	50	0.030000	NA	NA	1556360	7	16	NA	7
ns903616	21582	18052	25	0.900000	NA	NA	783520	22	41	NA	NA

Table 4: Table of next 21 instances.

Instance	#Vars	#Cons	Gecode Sol.	Init. Sol. Time	Feas.Time	Feas.Val	#Iter	Final Obj	Final Viol	Gurobi	Opt
opm2-z10-s2	6250	160633	100	0.750000	0.750000	0	1040	-9297	0	-24373	-33826
opm2-z11-s8	8019	223082	100	0.900000	0.900000	0	1097	-10921	0	-31311	-43485
opm2-z12-s14	10800	319508	100	1.520000	1.520000	0	1088	-14557	0	-43598	-64291
opm2-z12-s7	10800	319508	100	1.500000	1.500000	0	1408	-15398	0	-43139	-65514
opm2-z7-s2	2023	31798	100	0.110000	0.110000	0	4144	-3695	0	-10280	-10280
pb-simp-no	23848	1451912	100	5.300000	5.300000	97	523	97	0	140	NA
protfold	1835	2112	25	0.040000	NA	NA	880685	-14	3	-19	-31
p6b	462	5852	100	0.010000	0.010000	0	79578	-59	0	-63	-63
queens-30	900	960	100	0.090000	0.090000	0	3124	-38	0	-37	-40
ramos3	2187	2187	100	0.030000	0.030000	471	22563	258	0	255	NA
seymour	1372	4944	100	0.030000	0.030000	547	7780	446	0	424	423
sts405	405	27270	100	0.160000	0.160000	355	1786	347	0	347	NA
sts729	729	88452	100	0.490000	0.490000	661	253	652	0	651	NA
tanglegram1	34759	68342	100	2.320000	2.320000	7813	4899	7516	0	5182	5182
tanglegram2	4714	8980	100	0.080000	0.080000	2140	7175	2009	0	443	443
toll-like	2883	4408	100	0.030000	0.030000	1155	57839	931	0	627	610
t1717	73885	551	0	30.290000	NA	NA	96904	416368	25	238901	NA
t1722	36630	338	50	7.540000	NA	NA	81574	315879	33	136540	NA
vpphard2	199999	198450	0	2.990000	NA	NA	189071	752	28	81	81
vpphard	51471	47280	0	1.380000	NA	NA	294746	328	5	9	5
wnq-n100-mw	10000	656900	50	4.370000	10.18	5987	781	2411	0	284	259

Table 5: Table of last 21 instances.

9 Future Work

There is many things that can be developed to this framework. Find a way to incorporate Gecode such that is is easier to create new constraint. Currently it can only solve instances with integer coefficients because Gecode either takes all arguments as integer (and integer variables) or all arguments as float (and float variables). It has not been possible to scale the coefficient to integer since it quickly leads to integer overflow.
It can be extended to handle integer variables as well as binary. This could lead to a wide range of constraints to implement and increase the usability of Gecode. To handle integer variable new neighborhoods must be implemented and one should think about how to handle models with both binary and integer variables.
The NEIGBORHOOD classes is mixed with heuristics and it would be good to separate those, both conceptually and for effectiveness in practice.
There is currently not much a user can do to influence how the problem should be solved other than a priority for the constraints. It could be extended with a priority for the variables as well such that variables depend on other got a lower priority. In addition an option class could be implemented giving the user access to change some of the parameters in the framework.
Last but not least the parameters of the local search should be test more throughly which is expected to increase the performance of this solver.

References

[1] Horseshoe9. http://www.dcsc.sdu.dk/resources/horseshoe9/. [Online; accessed 31-01-2016].

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

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

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

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

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

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

[8] OscaR Team. OscaR: Scala in OR, 2012. Available from https://bitbucket.org/oscarlib/oscar.

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

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