A General Purpose Local Search Solver

January 21, 2016

Contents

1 Introduction **3**

2 Discrete Optimization **3**
- 2.1 Variables . . . 3
- 2.2 Constraints . . . 3
- 2.3 Problem Formulation . . . 3
- 2.4 Solutions . . . 4

3 General Purpose Solution Methods **5**
- 3.1 Binary- and Integer Linear Programming . . . 5
- 3.2 Constraint Programming . . . 5
  - 3.2.1 Implicit Constraints . . . 6
  - 3.2.2 Gecode . . . 6
- 3.3 Heuristics and Local Search . . . 7
  - 3.3.1 Basic Aspects of Local Search . . . 8
  - 3.3.2 Local Search Algorithms . . . 9
  - 3.3.3 Construction Heuristics . . . 9
  - 3.3.4 Metaheuristics . . . 10
  - 3.3.5 Creating a Model for Local Search . . . 10
- 3.4 Constraint Based Local Search . . . 11
  - 3.4.1 Invariants and Oneway Constraints . . . 11
  - 3.4.2 Soft Constraints . . . 11
  - 3.4.3 Evaluating Solutions . . . 12
  - 3.4.4 Comet . . . 12
  - 3.4.5 OscaR . . . 12

4 Architectural Overview **13**
- 4.1 Variables . . . 14
- 4.2 Constraints . . . 14
- 4.3 Invariants . . . 15
- 4.4 General Solver . . . 16

5 Preprocessing and Initial Solution **18**
5.1 Domain Reduction 18
5.2 Finding an Initial Solution 18

6 Structuring Local Search Model **20**
6.1 Simplification 20
6.2 Dependency Digraph 23
6.3 Propagation Queue 25

7 From Modeling to Local Search **27**

8 Local Search and Metaheuristics **28**
8.1 Neighborhoods 29
8.2 Search Procedures 32
8.3 Local Search Algorithms 34
8.4 Change in Local Search Design 37

9 Tests **37**
9.1 Gecode Search Engine 37

10 Results **37**

11 Future Work **37**

12 Conclusion **38**

1 Introduction

2 Discrete Optimization

2.1 Variables

Models contain a set of n variables $X = \{x_1, x_2, \dots, x_n\}$ and let $I = \{1, 2, \dots, i, \dots, n\}$ 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 $D = D_1 \times D_2 \times \dots \times D_n$ 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 $V(x)$. We say a variable is independent if the value is allowed to change within its domain in contrast to a dependent variable that only changes when other variable changes. We will denote with the letter y variables whose domain is the binary set $\{0, 1\}$.

2.2 Constraints

The values of variables will be restricted by a set of m constraints $C = \{c_1, c_2, \dots, c_m\}$ and let $J = \{1, 2, \dots, j, \dots, m\}$. The set of variables to which the constraint c_j applies is called its *scope* and is denoted $X(c_j) = \{x_{1j}, x_{2j}, \dots, x_{\alpha_j j}\}$. 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, $c_j \subseteq D(x_{i_1,j}) \times D(x_{i_2,j}) \times \dots \times D(x_{i_{\alpha_j},j})$, $x_{ij} \in X(c_j)$, 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
$\tau = (V(x_1), V(x_2), \dots, V(x_n))$ 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. If the restriction $\tau[X(c_j)]$ matches a tuble of the constraint c_j in extensional form the solution τ satisfies constriant c_j. 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 $f(\tau) \rightarrow \mathbb{R}$ that evaluates the quality of the solution τ, $\mathbb{P}' = \langle X, D, C, f) \rangle$. In the COP the task is then to find a feasible solution $\hat{\tau}$ in S such that $f(\hat{\tau}) \leq f(\tau)$ for all feasible solutions τ of $\mathbb{P}$ if it is a minimization problem.

3 General Purpose Solution Methods

3.1 Binary- and Integer Linear Programming

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

$$\begin{aligned} \text{Minimize} \quad & z = \mathbf{c}^T\mathbf{x} && (1) \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} && (2) \\ & \mathbf{x} \in \mathbb{Z}^n && (3) \end{aligned}$$

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

3.2 Constraint Programming

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

constraints posted. The language or framework may provides global constraint that can be used to formulate the problem. An example of a global constraint is the alldifferent(**x**) constraint that specify the variables **x** must have pairwise distinct values.
Two important aspects of solving a CSP are inference and search. *Inference* is adding constraints to the CSP that does not eliminate any feasible solution but might make it easier to solve the CSP [1, p.301]. Interference when dealing with variables with finite domain can local constraint propagation that is often used to eliminate large subspaces of the search space **(Search space not defined yet)**. Propagation can be restricting domains of variables, called filtering, or combinations of values to variabels, based on the constraint doing propagation [1, p. 169]. Propagation can be done when a constraint is created by eliminating possible assignments of the 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 $\{0, 1\}$.
(Tror jeg vil forklare om search space tidligere og introducere hvordan det bliver søgt her)
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 Implicit Constraints

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

3.2.2 Gecode

Gecode (generic constraint development environment) is a constraint programming solver implemented in C++ and offer a wide range of modeling features. Gecode offers more than 70 constraints from the “Global Constraint Catalog” [5] that can be applied to boolean, integer, set and float variables. **(Muligvis Gecode architecture billede)**
A model created for Gecode is created by inheriting the space class. Space is is a basic layer in Gecode that a user can build the model on. To Create variables or post constraints the user need to specify the space they should be created in. When variables are created in a space, views are created and associated with the variables. Views are not used in modeling but are used to know when propagation should be made on a constraint. When posting

constraints in a space, Gecode creates propagators and these propagators can subscribe to the views of the variables in the constraint. When variables changes domain the corresponding view tell its subscribes that the variables domain has changed. For some constraint the user has the option to choose the propagator based on a consistency level. The cost of different consistency level varies from linear in the number of variables to exponential [6, p.57].
To solve a problem Gecode needs guidance when searching and that is done by a branch function. Once a problem has been formulated, the user must define on which variables and how branching is done. Just like variables and constraints are posted in a space the branch order is also posted on the space. The choices in branching for a set of variables are which of those variables to branch on first and what values to branch on. One can post several branch methods and they are treated in the order they are posted. Once all variables have been branched in one branch function it continues with the If no branch strategy is chosen for a variable then branching is not done on that variable.
To start the search a 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 [6, 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 solution is possible it stop branches from that vertex and the space is said to be failed. While searching for solution sometimes, based on a search parameter from the option given, Gecode clones the spaces. When Gecode reaches a failed space, instead of starting from scratch and recompute all way to down to the previous vertex, it uses the closest clone to backtrack to that space.

3.3 Heuristics and Local Search

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

3.3.1 Basic Aspects of Local Search

There are several important aspects of local search and the most basic will be covered here.
The evaluation function $f(\tau)$ evaluate the quality of the candidate solution. The *step function* that defines the candidate solutions close to a given candidate solution τ. By applying the step function once to candidate solution τ a new candidate solution τ' is obtained. Going from one to solution to another with the step function is called a *step*. Several solution might be explored before making a step and using a *neighborhood operation* defines by the step function. Local search uses a neighborhood function $N(\tau)$ that for each solution $\tau \in \S$ specify a subset of solutions $N(\tau) = \mathcal{T}$. The solutions $N(\tau)$ is called the neighborhood of τ and are the set of solution that can obtained by making one step. The cardinality of $N(\tau)$ is called the neighborhood size of τ and depends on the step function. The neighborhood function N is symmetric if, $\tau' \subseteq N(\tau)$ if and only if $\tau \subseteq N(\tau')$ for all pairs of τ and τ'. In a problem with n binary variables the step function could be to change the value of a single variable, a flip. This can be done on all variable hence the neighborhood size of a candidate solution would be n.
It might be expensive to use the evaluation function to evaluate each solutions, instead a *delta evaluation function* $\delta(\tau)$ can be used. The evaluation function recomputes the quality of a solution even if only a few variables changes value. The delta evaluation function only computes how much the 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 S and E is a set of edges connecting a vertex v, representing the solution τ, with the vertices representing $N(\tau)$. The graph G is called the neighborhood graph and local search does a walk through the neighborhood graph when searching for a solution. [4, p. 3-5]
Local search needs a *termination criteria* that determines when the search should stop. Sometimes we know what an optimal is, i.e. the SAT problem, once we find a feasible solution we can stop. In other cases an optimal solution may not be know, several combinatorial problems are not solved to optimality. The termination function can i.e. be based on a time limit, number of steps made, or when a locally optimal solution is found. A locally optimal solution for a minimization problem is a solution $\hat{\tau}$, such that for each feasible solution $\tau \in N(\hat{\tau})$ $f(\hat{\tau}) \leq f(\tau)$.

3.3.2 Local Search Algorithms

When a local search algorithms makes a move from a current solution to new solution we say it commits to the new solution. 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 move to make.
Two basic pivoting rules for iterative improvement are *Best improvement* and *first improvement.* Best improvement examine 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 neighborhood and commits to the first solution that gives an improvement. 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 to one of the neighbor solution chosen uniformly random. This might lead to worsening solution and/or infeasible solution and is usually combined with other heuristics or local search algorithms.
Before local search can be done an initial assignment to the variables is needed and an initialization function is used for this also called a construction heuristic.

3.3.3 Construction Heuristics

A construction heuristics is used to find an initial candidate solution to a given instance. One of the main things to consider when creating a construction heuristic algorithm is the balance between quality of the solution and time complexity of the algorithm. The extreme case would be to solve the given instance with a construction heuristic but then the main point of local search is lost, that is finding a high quality solution fast.
The first thing to consider when creating a construction heuristic is whether it has to find a feasible solution or it is allowed to find an infeasible solution. The choice depends on how the local search is designed, whether it can reach infeasible solution or not.
An example of a simple construction heuristic is creating a random candidate solution. For each variable a value between its lower and upper bound is chosen uniformly at random. This is a fast construction heuristic $O(n)$

but cannot give any guarantee of the quality. Examples of other construction heuristics can be greedy heuristics like first fit or best fit.
Construction heuristics can be tailor made to a specific problem type to find a good initial solution depending on the problem. It might be beneficial to introduce randomness in the algorithm and rerun it a couple of times and to get a better initial solution at the cost of time.

3.3.4 Metaheuristics

Metaheuristics defines how the search space should be explored in where as local search algorithms that focus more on the neighborhood of the solution. Iterative improvement is often fast to find a local optimum depending on the size of the neighborhood. Usually only a small fraction of local optima are close to optimality and they can be a poor quality solution [4, p.135]. Metaheuristics are used to get out of local optima to search different parts of the search space.
One of the simple Metaheuristics is *iterative local search* that remembers the best solution found so far and uses two local search algorithms, random walk and iterative improvement. It uses iterative improvement to find a local optima and compare it to the best solution so far. Then uses random walk for some iterations to escape the local optima. These two algorithms are repeated until the termination criteria is reached.
Another metaheuristics is *tabu search* that implements an iterative improvement with a modified best improvement pivoting rule. It chooses the solution leading to the best solution but not necessarily an improving move. In addition to the modification of the pivot rule it implements a tabu list T that remembers the last m solution. The solution in the tabu list are not consider when looking for the next solution. It might be very memory expensive to keep track of the last m solutions. An alternative way of implementing the tabu list is to remember the last m moves and forbid the reverse move. This is more restrictive since the reverse moves might not lead to a solution visited within the last m iterations. To compensate for this a *aspiration criteria* can be implemented. Aspiration criteria is a set of rules that can overrule the tabu list [4, p.139-140].
Several other metaheuristics exist such as variable neighborhood search and very large scale neighborhood search. Variable neighborhood search uses different step function, hence different neighborhoods. Very large scale neighborhood search changes many variable in each move which give a very large neighborhood but might need fewer iteration to find a good quality solution.

3.3.5 Creating a Model for Local Search

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

3.4 Constraint Based Local Search

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

3.4.1 Invariants and Oneway Constraints

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

$$z = f(\mathbf{X}) \tag{4}$$

Where $f(\mathbf{X})$ is a oneway constraint with a set of input variables X that functionally defines the invariant z.

3.4.2 Soft Constraints

Soft constraints are constraints that the CBLS should try to satisfy when making moves. 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 violation in a constraint. One way is a boolean value, zero for not violated and one for violated. Another way is measure how much a constraint is violated, *violation degree*, by a function depending on the constraint. I.e. for the **alldifferent(X)** constraint, that is all variables **X** must have pairwise distinct values, it can be the number of variables with the same value as another variable.

3.4.3 Evaluating Solutions

There are different ways of measuring the quality of an solution. For a CSP a feasible solution needs to be found but there is not distinguished between feasible solutions.
In a COP the feasible solutions are compared by an objective function that gives the feasible solution a measurement of quality.
There are different ways of differentiating infeasible solution from other solutions, feasible or infeasible. The evaluation function is a combination of the objective function and the violated constraints. Each constraint can be given a weight and together with the violation degree it contribute to the evaluation function. I.e. a constraint has weight ten and a violation degree of three it then contribute 30 to the evaluation function. In this way the search priorities to satisfy certain constraints more than other. This method can have the negative effect that it might priorities to have all constraint with violation degree over one constraint with very high violation degree.
Another way it to use a priority for each constraint as a lexicographic ordering, rather satisfy a constraint with high priority than any number of constraints with lower priority.

3.4.4 Comet

3.4.5 OscaR

4 Architectural Overview

This section gives an overview of the key components of the solver. The figure 1 gives an overview of the most basic classes and the classes they have pointers to (access to).

The two engines for solving are the Gecode Engine and Local Search Engine that find the initial solution and optimize the solution respectively. Gecode Engine is used for preprocessing and finding an initial solution, if possible with in the limits given. **(Either just time limit or could be made visible to the user by an option class with node, fail, and time limit.)** Gecode Engine will be elaborated further in section 5.

Local Search Engine is responsible for the optimization part of the solver with the use of local search and metaheuristics. How the optimization is done is described in section 8. Local Search Engine transform the model to a model better suited for local search before the local search can start. How this is done and why will be discussed in section 6.

The Model class contains pointers to components of a CBLS model, variables, constraints, and invariants. The engines naturally has access to these objects and Local Search Engine can add new objects such as invariants.

Constraint and Invariant are super classes to all constraint and invariant respectively and are described in subsection 4.2 and 4.3. They contain abstract methods that the subclasses must specified.

The main part of the solver is the General Solver class that contains the engines used for solving. The General Solver class contains the methods that are called by the user, such as creating variables and constraints, finding initial solution and optimizing the solution discussed in subsection 4.4.

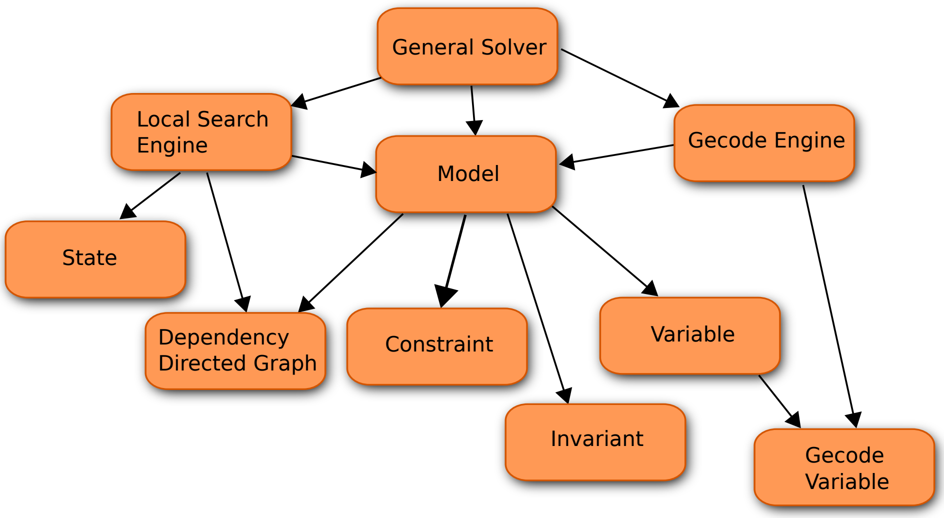

Figure 1: Overview of what the classes contains pointers to

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

4.1 Variables

4.2 Constraints

Constraints are all derived from the same class (Constraint figure 2) that forces some methods to be implemented. All constraints need a priority according to how important the constraint is, that is an integer. The priority do not need to be different for the constraints but it will help the local search to differentiate between infeasible solutions.
The method getType is only used if Gecode does not find an initial solution. How Gecode is used is discussed in section 5.
An example of a constraint is the Linear constraint, which is the same as the one used in Integer and binary programming, equation 4.2 is an example of a Linear constraint.

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

A constraint is posted in the constraint programming environment and later handled in the local search environment. The constraints are treated

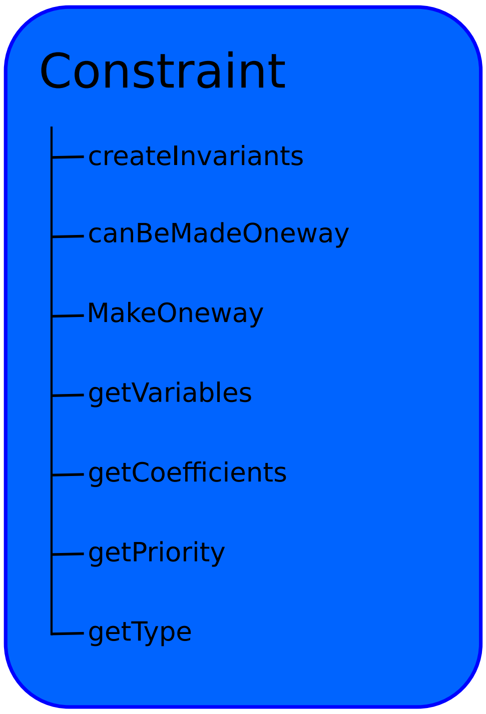

Figure 2: Important methods in Constraint class

differently in the environments and need different parameters and methods for that. The LS environment handles constraints through invariants hence a constraint needs a method for creating the invariants needed in LS for the specific type of constraint. The method createInvariants creates auxiliary variables, invariants, that might be auxiliary variables helpful during local search and one invariant that represents if the constraint is violated and the cost of being violated. The violation cost can be one if violated and zero otherwise, or it give a measurement of violated the constraint is. For the linear constraint c_1 in the example the measurement could be how much the left hand side should 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 returns the oneway constraint, implemented as invariant, that defines a variable. For the Linear constraint only the constraints with an equal relation can be used, hence not c_1. Why this is done and an example of how is described in section 6.

4.3 Invariants

Invariants are all derived from one common class Invariant, figure 3, for the same reason as constraints are. Invariants are only introduced after an initial solution to an instance has been found and before the local search has

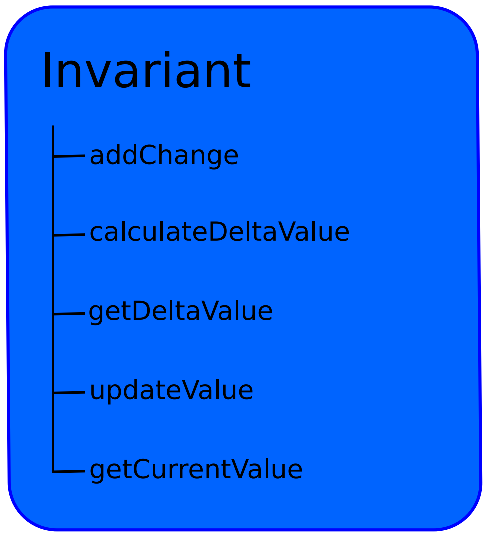

Figure 3: Important methods in Invariant class

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. This mean the invariant classes are representing both the oneway constraint and the invariant.
All Invariants must implement the methods proposeChange, calculateDelta , and updateValue that are used during local search. The proposeChange is used to tell an invariant that a variable in the oneway constraint defining it has been changed. The 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 move is performed 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.
Some of the classes created in the Local Search Engine uses invariants but do not differentiate between them. 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 is implemented by the new invariant and nothing has to be changed in the Local Search Engine.

4.4 General Solver

The General Solver 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 Gecode Engine and variables used in the Local Search Engine class. The variables used in the Local Search Engine class (LS variables) are of the class Variable and each has a pointer to the associated Gecode variable. The method returns pointers to the LS variables.
The method relax is only used if an initial solution could not be found within the limits given. It controls how the instance should be relaxed to find an initial solution. There is currently only one method implemented that relaxes the an instance that is described in subsection 5.2.
All constraint available are created by calling the associated method in General Solver , that calls the constructor of the constraint and the method in Gecode Engine for posting the constraint in a Gecode space. The constraint

objects should not be created directly by the user since these do not post the related constraint in the Gecode space.
To implements a new constraint object it must inheriting from the Constraint class and two methods, one in General Solver and one in Gecode Engine, must be implemented. The method in Gecode Engine must post the constraint in the Gecode Engine space. The method in General Solver should call the constructor of the constraint implemented and call the method in Gecode Engine. The solver must be able to reproduce the call to Gecode Engine in case it an initial solution is not found within the limits given. The relaxation method must be updated to handle the new constraint implemented as well. **(Could avoid inserting code in methods if constraints have pointers to the Gecode space but conceptually it does not make sense.)**
To find an initial solution the method initialSolution must be called and it takes an integer argument. The argument indicates the time Gecode is allowed to search for an initial solution before relax is called. Once relax is called the same time limit is given again.
To find a better solution than the initial solution the method optimizeSolution can be called with a time limit as argument. That method starts the local search that is described in section 8.

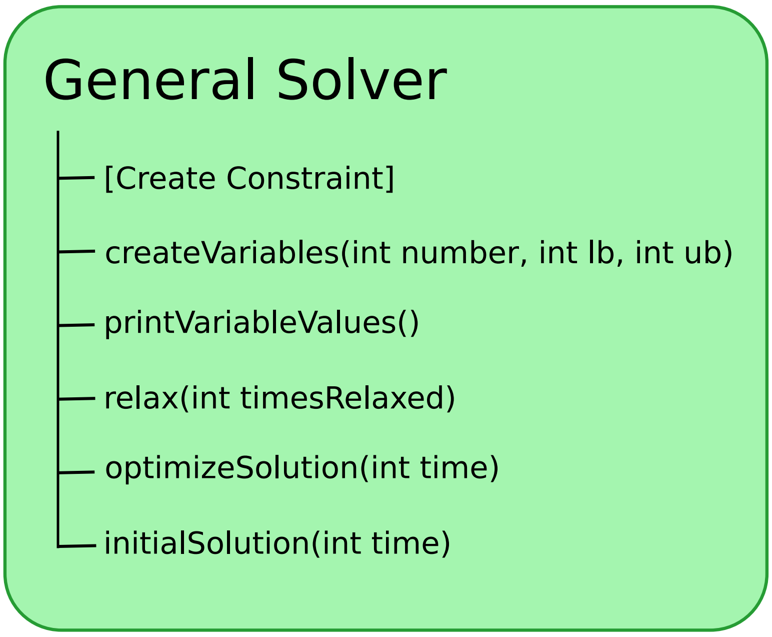
General Solver
[Create Constraint]
createVariables(int number, int lb, int ub)
printVariableValues()
relax(int timesRelaxed)
optimizeSolution(int time)
initialSolution(int time)

Figure 4: Important methods in General Solver class

5 Preprocessing and Initial Solution

Each time a constraint is posted by a user it is parsed from General Solver to the Gecode Solver. The Gecode solver is derived from Gecode Space **(Define structure / Space before this)** and the constraints parsed is posted in the Gecode Solver. Gecode has a large selection of constraints that can be used [6, p. 58-80]. Most of the constructors of these constraints are overloaded such that they can take different arguments and works with different types of variables.

5.1 Domain Reduction

When an initial solution to the model is requested, Gecode Branch method is called that specifies which variables to branch on and what values should be examined in the branches. Gecode uses preprocessing before searching for a solution and that might reduce the domain of some of the variables. If the domain of a variable is reduced to a single value, the variable is said to be fixed and is assigned that value.

5.2 Finding an Initial Solution

Once domain reduction preprocessing has been done a Gecode DFS search engine is started. The stop criteria for Gecodes search can be specified by an option class. A Gecode search engine takes a space and search option as arguments and the search option contains a stop object. The stop object can either be timestop, nodestop or failstop. Each time Gecode branches on a variable two new nodes are created and nodestop set a upper bound on the number of nodes to explorer. If Gecode reaches a node that gives an infeasible assignment to a variable 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 having only one of these stop object a multistop object has been created that combines all three stop objects if the user wants to having multiple stop criteria.
Combinatorial problems can be formulated with Gecode and these problems can be very difficult to solve. In these cases Gecode keeps searching for a solution until it finds one (or runs out of available memory). Instead, the search can be stopped using stop object and the constraints can be relaxed such that Gecode can find a initial solution to the instance. If one of the stop criteria is reached the relax is called and some of the constraint are relaxed. To relax some constraints a new Gecode Engine is created and all variables and constraints, except those relaxed, are created and added to the new space. In order to choose which constraint that should be removed the priority of the constraints, given by the user, is used. Different techniques can

be applied for choosing which constraints with the same priority to remove. The one chosen here is a stepwise backward algorithm, that is a greedy approach. **(Article Marco talked about propose some method, but says a greedy approach is good as well?)**. The number of constraints that are relaxed/removed is based on the number of constraints $|C|$ and the number of restarts made r. The number of relaxed constraints is given by equation (6).

$$Relaxed = \left\lceil \frac{2^r}{100} \cdot |C| \right\rceil \tag{6}$$

After six relaxations 64 % of the constraints have been relaxed and if no solution can be found within the search limits, the search is stopped and finding a solution to the instance has failed.
The constraints are chosen by their priority and ties are broken at random. I.e. a model with 100 constraint of priority 1, 40 of priority 2, and 15 of priority 3 and there are 20 constraints that should be relaxed. The 15 constraints with priority 3 and 5 of the 40 constraints with priority 2 are chosen at random would not be posted. The constraints that are not posted in the Gecode space are still applied when doing local search, hence the initial solution might start with some violations.

6 Structuring Local Search Model

Once an initial solution to the constraint satisfaction problem (CSP) has been found by Gecode the model is transformed to create a model suited for local search, a CBLS model. Two new datastructures structures are introduced in this section, dependency directed graph in subsection 6.2 and propagation queue in subsection 6.3.
The dependency directed graph is used to update invariants when a variable changes value. A propagation queue q_i is created for each variable x_i that gives an ordering of the invariants reachable from x_i in the dependency directed graph.

The model is simplified by defining some of the variables by transforming the functional constraints into oneway constraints using two algorithms described in subsection 6.2. When a variable is defined by a oneway constraint it is transformed into an invariant since its value is dependent on other variables and invariants.

6.1 Simplification

For each functional constraint c_j two algorithm steps are used to create invariants, one 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 1: canBeMadeOneway(Constraint c_j)

```
input : Functional constraint $c_j$
1  Variable $bestVariable$ = NULL
2  numberOfTies = 0
3  foreach $x_i$ in $X(c_j)$ do
4      if $x_i$ is fixed or defined then
5          continue
6      end
7      // Break ties
8      if defines($x_i$) < defines($bestVariable$) then
9          // Choose the variable that helps define fewest
               invariants
10         $bestVariable = x_i$
11         numberOfTies = 0
12     end
13     else if defines($x_i$) == defines($bestVariable$) then
14         if $|D(x_i)| > |D(bestVariable)|$ then
15             // Choose the variable with largest domain
                   size
16             $bestVariable = x_i$
17             numberOfTies = 0
18         end
19         else if $|D(x_i)| == |D(bestVariable)|$ then
20             if $|deg(x_i)| < |deg(bestVariable)|$ then
21                 // Choose the variable with lowest degree
22                 (remember to define degree) $bestVariable = x_i$
23                 numberOfTies = 0
24             end
25             else if $|deg(x_i)| == |deg(bestVariable)|$ then
26                 // Fair random
27                 numberOfTies++
28                 if Random(0,numberOfTies) == 0 then
29                     $bestVariable = x_i$
30                 end
31             end
32         end
33     end
34 end
35 if $bestVariable \neq$ NULL then
36     makeOneway(Constraint $c_j$, Variable $bestVariable$)
37 end
```

For each functional constraint a non-fixed and non-defined variable is found if possible. If there is more than one eligible variable the best variable among those is found. The first tiebreaker is the number of oneway constraints the variable participate in (helps define other variables). The next tiebreaker is on the domain of the variables, the third is the number of constraints the variables participate in. If none of the tiebreakers can be used a fair random is used such that the probability is equal for all variables whose ties could not be broken.
Once a the best variable is found, if any, the algorithm 2 `makeOneway` is called.

Algorithm 2: makeOneway(Constraint c_j, Variable x_i)

```
input  : Constraint $c_j$ and Variable $x_i$
output: Updated $G$

1  set $Q$                                  // new coefficient set
2  set $U$                                    // new variable set
3  // Move $x_i$ to right hand side and set coefficient to 1
4  foreach $x_k$ in $X(c_j) \setminus x_i$ do
5      $c'_{kj} = -\frac{c_{kj}}{c_{ij}}$
6      $Q = Q \cup c'_{kj}$
7      $U = Q \cup x_k$
8  end
9  // Move right hand side to left hand side and update
      coefficient
10 double $b' = \frac{b(c_j)}{c_{ij}}$
11 (Remember to define $b(c_j)$ before this) (coefficients can
   now be doubles (non integer))
12 invariant $inv$ = new Sum ($U$,$Q$,$b'$)
13 // Invariant that has the value (the sum of) the left
      hand side
```

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

For each of the remaining constraints in G auxiliary variables are introduced as invariants. In figure 6 the invariant i_2 is an example of an auxiliary variable. The value of the invariant is the value of the left hand side of the corresponding constraint. These invariants are used to speed up local search, that is described in section 8.

6.2 Dependency Digraph

The dependency directed graph (DDG) $G = (V, A)$ is made of a set of vertices V representing the non-fixed variables, invariants and the non transformed constraints. The vertex $v \in V$ has an outgoing arc to vertex $u \in V$ if and only if the value of the variable corresponding to u is directly dependent on the value of variable corresponding to v. The variable vertices only have outgoing arcs and the constraints can only have ingoing arcs. **(Skal det stå her efter omstrukturering)** The initial model will be modified by introducing invariants defined by oneway constraints and vertices representing invariants will be added to the graph G.
The graph can be illustrated with all the variable vertices to the left with outgoing arcs going right to vertices representing invariants and constraints. The invariants are variables that are defined by oneway constraints or they can be auxiliary variables used in the local search. If a variable is defined by a oneway constraint the variable vertex is removed from G since the value of that variable is given by the invariant representing it.
The DDG is used to update values of variables and invariants during local search. The graph G is used to build the propagation queues described in subsection 6.3.
The example is a model with three variables and a two constraint and will illustrate how a possible dependency directed graph G is made.

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

G consist of the three variables x_1, x_2, and x_3 and the constraints c_1 and c_2. The variable x_3 can be defined as an invariant inv_1 by transforming c_1 to a oneway constraint. Once variable x_3 is defined by a oneway constraint c_1 and x_3 are removed from the graph and replaced by invariant inv_1. The variables x_1 and x_2 defines inv_1 hence they have outgoing arcs to inv_1. Invariant inv_1 has an outgoing arc to c_2 since Variable x_3 participates in c_2.
Auxiliary variable can be useful to update constraint violations and in this example we could create an auxiliary variable where value is the sum of the left hand side of c_2. The auxiliary variable will be represented by an

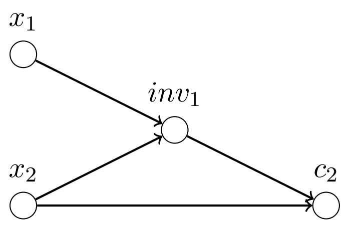

Figure 5: Small example of DDG

invariant inv_2 which will be added to G. The invariant inv_1, representing x_3, and variable x_2 have an outgoing arc to inv_2 that has an outgoing arc to c_2. When changing the value of x_2 both invariants need to be updated since they are dependent on the value of x_2. Invariant inv_2 is dependent on the value of inv_1 therefore to avoid updating inv_2 twice, it is beneficial to update inv_1 before updating inv_2. This is the ordering given the propagation queue that is discussed in the next subsection.

In order to avoid circular definitions of invariants dependency directed graph G should be acyclic. A circular definition could be if x_i is used to define x_j and vise versa. Then a change in value of x_i would lead to a change in value of x_j that again changes the value of x_i and so on.
Once all invariants are introduced, 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 [2, p. 1170]. To find all SCC of size two or more Tarjan's algorithm **(cite SIAM J. Comput., 1(2), 146–160.)** that finds strongly connected components (SCC) is used. **(Description of Tarjan and timestamps)**
Each of these strongly connected components must be broken in order to keep G acyclic, since a SCC consist of at least one cycle. A SCC can be broken by removing arcs and/or removing vertices. The arcs A represent relations between variables, invariants and constraints and should not be changed. The vertices V^{SCC} can only be vertices representing invariant since variable only have outgoing arcs and constraint only have ingoing arcs. Undefining one of those invariants corresponds to removing one of the vertices, hence breaking the SCC. An invariant can be undefined by reintroducing the variable it represents and removing the invariant from the model. The oneway constraint used to define the invariant is transformed back into a functional constraint again and is reintroduced in the model.
For each SCC one of the vertices is chosen and the invariant it represents is undefined. The invariant is chosen in the order of lowest domain then highest arity of oneway constraint defining it **(Search priority?)**. If there is ties they are broken at random.

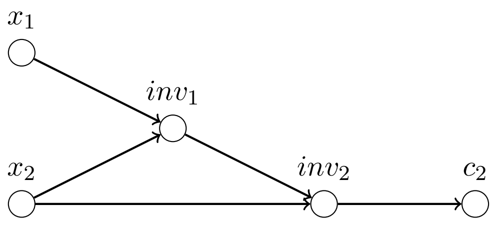

Figure 6: Small example of DDG continued

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 first tiebreaker in algorithm 2 is used as a heuristic to reduce the number of cycles generated. If invariants are not used to define other invariants no cycles can occur, since cycles can only be made of invariant vertices.
Tarjans algorithm also gives each vertex representing invariants a time stamp that is used to create the propagation queues described in the next subsection. **(This should be described earlier)**

6.3 Propagation Queue

For each independent variable x_i a propagation queue q_i is made. A propagation queue q_i is an topological sorting of invariants that are reachable from the vertex representing x_i in the dependency directed graph G **(maybe define topo sort)**. The propagation queue q_i is used such that each invariant dependent on the value of x_i is updated at most once if the variable changes value. The DDG represents which invariant that are directly affected by a change in variable x_i but not the order in which they should be updated. Figure 6.3 shows the necessity of such an ordering.
If inv_1 is updated before inv_2 then it might need to be updated again after inv_2 is updated hence updated twice. In worst case the number of updates performed when updating x_i could be exponential in the number of vertices reachable from x_i instead of linear.
Once the dependency directed graph is a DAG each invariant vertex has been given a time stamp by Tarjans algorithm. Propagation queues are implemented as red-black trees without duplicates hence they have insert time complexity $O(log(n))$. For each variable vertex x_i in dependency digraph G a depth first search is made and each vertex visited is added to the propagation queue of x_i **(currently revisits vertices and adding vertices**

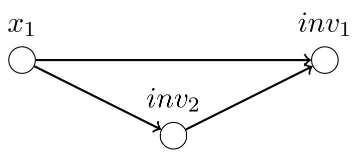

Figure 7: Importance of propagation queue

pointed to agian. Should be fixed). The vertices in the propagation queue are ordered according to their time stamp in decreasing order which is a topological sorting such that there is no backward pointing arc.

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

7 From Modeling to Local Search

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

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

Table 1: Suggested activities

8 Local Search and Metaheuristics

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

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

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

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

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

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

Each constraint $c \in C$ created in the model was given a priority p and these priorities are used during local search and let k denote the highest priority given. **(Define $V(c)$, violation degree and $P(c) > 0$ priority of c).** Let $q_p \in Q$ be the sum of violation degree $V(c)$ of constraints with priority p. The evaluation functions value is consider as q_0 and the vector Q is used to evaluate the quality of a solution. Two candidate solutions τ and τ' each have a vector of quality Q_{τ} and $Q_{\tau'}$ respectively. To determine which of the two solution are best their vector can be compared, starting with position k and going backwards. The first position they have different value determine which solution is best, the one with the lowest value. Illustrated with a small example:

$$Q_{\tau} = (5, 2, 4, 2) \tag{7}$$
$$Q_{\tau'} = (10, 6, 3, 2) \tag{8}$$

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

8.1 Neighborhoods

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

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

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

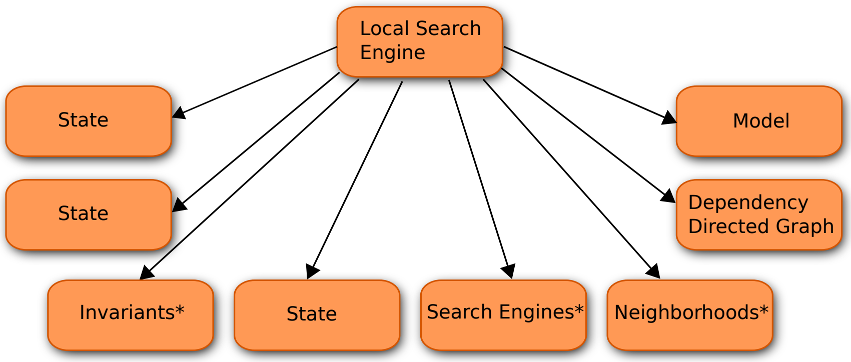

Figure 8: Overview of the class pointers of Local Search Engine. The fields marked with a star (*) are several classes of that type.

(Variable og constraint til venstre, think this should be posted earlier)

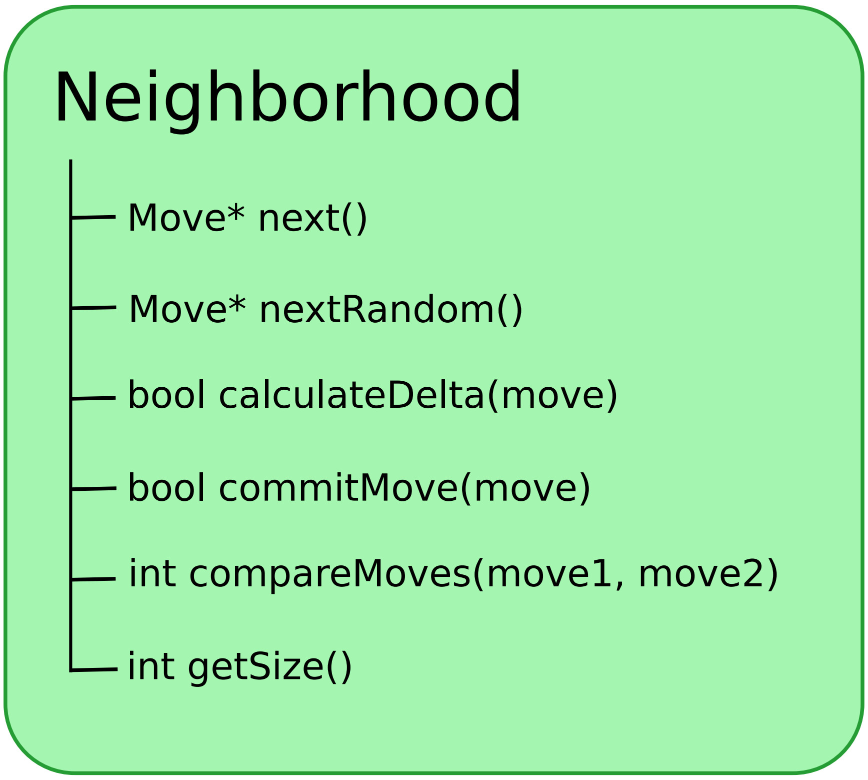

Figure 9: The methods all Neighborhood classes needs to implement.

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

Table 2: Table of Neighborhood classes

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

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

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

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

8.2 Search Procedures

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

The class BestImprovement looks at each Move a Neighborhood class NE gives and finds the best Move. The best Move is determined from their delta vector after the method claculatDelta of the Neighborhood is called on each Move. It returns a boolean that tells if the current solution was improved. A boolean can be given to BestImprovement that indicate if it should commit a non improving Move. How each iteration is done is describe by algorithm 3

Algorithm 3: BestImprovement Start

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

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

If BestImprovement is combined with the neighborhood class RandomConflictConNE it gives a minimim conflict heuristic that can be useful to reach a feasible solution.
FirstImprovement has a very similar implementation. Instead of calculating each Move of a Neighborhood class NE it stops requesting a Move once an improving Move is found. If no improving Move is found when NE returns a **NULL** pointer it does not commit a Move. If no improving Move is found the current solution is in a local optima with the regard to the chosen Neighborhood.
The class RandomWalk uses the method nextRandom() from its Neighborhood NE and if that Move is allowed it is committed. It takes an integer as argument that indicate the number of times it is repeated. The benefits is create new solution very fast but they are not likely to have a good quality.
Though tabu search is a metaheuristic it is implemented the same way as the other search procedures but with some additions. It takes four arguments; the number of steps made(iterations), the best solution found, the current solution, and a tabu list. The implementation is similar to BestImprovement with additional checks with regard to the tabu list and aspiration criteria. If a neighborhood operation is tabu but leads to a solution better than one found so far, the tabu list is ignored. The algorithm for tabu search for a single flip neighborhood sketched by algorithm 4.

Algorithm 4: 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.

8.3 Local Search Algorithms

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

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

Algorithm 5: Local Search - Test Algorithm 1

```
ConflictOnlyNE neigborhood
RestrictedFlipNE neigborhood2
TabuSearch TSCON(neighborhood)
TabuSearch TSRFN(neighborhood2)
int iteration = 0
int [] best = getSolution()
int [] current = getSolution()
int [] tabulist(neighborhood.getSize(), -neighborhood.getSize())
while within time limit do
    if Current solution is feasible then
        TSCON.start(iteration ,current, best, tabulist)
        iterations++
        if getSolution() is better than bestSolution then
            bestSolution = getSolution()
        end
    else
        TSRFN.start(iteration, current, best, tabulist)
        iterations++
        if getSolution() is better than bestSolution then
            bestSolution = getSolution()
        end
    end
end
```

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

Algorithm 6: Local Search - Test Algorithm 2

```
FlipNeighborhood FN
int randomMoves = min(FN.getSize() / 50, 10)
FirstImprovement FI(FN)
RandomWalk RW(FN, randomMoves)
bool improvement = true
while within time limit do
    if improvement && within time limit then
        improvement = FI.start()
        iterations++
        if getSolution() is better than bestSolution then
            bestSolution = getSolution()
        end
    else
        R
    end
    W.start()
    iterations += randomMoves
    if getSolution() is better than bestSolution then
        bestSolution = getSolution()
    end
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 7: Local Search - Test Algorithm 3

```
RandomConflictConNE neigborhood
RestrictedFlipNE neigborhood2
BestImprovement BIRCC(neighborhood)
TabuSearch TSRFN(neighborhood2)
int iteration = 0
int [] best = getSolution()
int [] current = getSolution()
int [] tabulist(neighborhood.getSize(), -neighborhood.getSize())
while within time limit do
    if Current solution is feasible then
        BIRCC.start()
        iterations++
        if getSolution() is better than bestSolution then
            bestSolution = getSolution()
        end
    else
        TSRFN.start(iteration, current, best, tabulist)
        iterations++
        if getSolution() is better than bestSolution then
            bestSolution = getSolution()
        end
    end
end
```

8.4 Change in Local Search Design

9 Tests

9.1 Gecode Search Engine

10 Results

11 Future Work

(Short description of problems that should be investigated more)
(preprocessing :) preprocessing of Cplex and gurobi with Gecode
(Cycles :) Finding all (elementary) cycles in the dependency digraph and/or finding the smallest set of vertices to remove such that the graph is DAG.
(Integer variables :) How to treat integer variables when they cannot be defined by oneway constraints.

(Propagation queue :) Finding a datastructure better suited for propagation queues than red-black trees (C++ set).
(Mixed neighborhood :) Find a way to treat both integer and binary variables in local search.
(Make "true" DFS when creating propagation queues, no revisit of vertices)
(Change Constraints to invariants during local search to measure the violation)
(option class for user to specify stop limits, branch other things)

12 Conclusion

References

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

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

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

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

[5] Sophie Demassey Nicolas Beldiceanu.

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