# A General Purpose Local Search Solver

September 22, 2015

## 1 Local Search and Neighbourhoods

## 2 Heuristics and Metaheuristics

## 3 Defenitions of structures in a solver

structure intro

### 3.1 Invariant

Invariants are variables, which value is purely determined by other variables. The variable defined is not necessarily one of the variable currently in the system but it can be a new auxiliary variable which value is of interest. The invariants all have generic methods that needs to be defined for the different types of invariants. One of these methods is for instance getCurrentValue() that return the value of the invariant. If a variable $v$ changes value then the invariants that are defined by $v$ needs to update their current value. This is done through the addChange(arguments) method, calculateDeltaValue(), and updateValue(). They informs the invariant something is changed, calculates the change in value and updates the current value respectively. Since invariants can be defined by variables that are invariants themselves this can leads to a series of updates. Invariants and constraints can built as a directed graph without cycles in order to avoid looking at an invariant multiple times when a change is made to a variable. The construction of that graph is described in section 5.1.
An example of an invariant is the Sum invariant that defines an auxiliary variable. The Sum invariant consist of a coefficient set $C$, variable set $V$ and it can have a constant $b$ added. For a Sum invariant $S$ the value is defined as:

$$S = b + \sum_{i \in I_S} c_i x_i \tag{1}$$

### 3.2 Oneway Constraints

One-way constraints are constraints that is used to define the value of a single variable. The single variable is made as an invariant and can only change when one of the variables that defines it changes. One-way constraints reduces the search space (which should be discussed before this I guess) since the variable defined by a one-way constraint can only be changed by a move indirectly.

# 4 Existing Solvers

## 4.1 Comet

Comet is an object orientated 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 language is the incremental store that contains various incremental objects fx. incremental variables. Invariants, also called one-way constraints, are expressions that are defined by incremental variables and a relation of those. An incremental variable $v$ can for instance be expressed as a sum of other variables. The variable $v$ will automatically be updated if one of the other variables changes value. The order in which the invariants gets updated can be implemented to achieve higher performance.
One layer above the invariants is the differentiable objects that can use the invariants and the incremental objects. Both constraints and objective are implemented as differentiable objects. They are called differentiable because it is possible to compute how the change of a variable value will affect the differentiable object's values. All constraints are implemented using the same interface, that means that all constraint have some methods in common. These method is defined as invariants hence they are always updated when a change is made to one of their variables. This is especially useful when combining multiple constraints in a constraint system, that also implements the constraint interface. The constraints can be combined in a constraint system that then uses the method from the individual constraints to calculated its own methods. Just like the `constraint` interface there exist a `objective` interface. Both will be described further in section 4.1.2.
The next layer is where the user models their problem and use the objects mentioned above. Several search method are implemented that can be used. The benefit of this architecture is the user can focus on modeling the problem efficiently on a high level and thereby avoid small implementation mistakes. Using constraint programming inspired structure gives the benefit of brief but very descriptive code.

### 4.1.1 Invariants

### 4.1.2 Differentiable Objects

Comets core modeling object is the differentiable objects that are used to model constraints and objective functions.
All constraints in Comet implement the possible to combine the constraints and reuse them easily. A constraint has at least an array $a$ of variables as argument, that is the variables associated with the constraint. The interface specifies some basic methods that all constraints must implement such as `violations()` that returns the number of violations for that constraint and `isTrue()` that returns whether the constraint is satisfied. How the number of violations is calculated depends on the implementation of the constraint. An example could be the `alldifferent(a)` constraint, that states that all variables in the array $a$ should have distinct values. The method `violations()` then returns the number of variables that do not have a unique value.

One of the benefits that all constraints implements the same interface is the ease of combining these constraints. This can be done by using other differentiable objects such as logical combinators. These objects has a specific definition of the basic methods from the Constraint interface that means they do not rely on the semantics of the constraints to combine them. Another way to combine the constraints is to use a constraint system. Constraint systems are containers objects that contains any number of constraints. These constraints are linked by logical combinators, namely conjunction. Comet can use several constraints system simultaneously and is very useful for local search where one system can be used for hard constraints another for soft constraints. Other constraint operators in comet are cardinality constraints, weighted constraints and satisfactions constraints.

Objectives are another type of differentiable objects and the structure is very close to the structure of constraints. Objectives implements an interface Objective that have some of the same methods as Constraint but instead of having isTrue() and violations () they have value() and evaluation(). The method value returns the value of the objective but it can be more convenient to look at the method evaluation() to guide the search. The method evaluation() should indicate how close the current assignment of variables is to improve the value of the objective or be used to compare two assignments of variables. How the evaluation should do that depends on the nature of the problem and could for instance use a function that would decrease as the assignment of variables gets close to improve the value().

### 4.2 Gecode

bla3

### 4.3 Localsolver

bla2

## 5 Graph

There can be many dependencies between variables,invariants and constraints and these dependencies can be illustrated by a graph. The graph can be partioned into to parts. The first part consist of a vertex for each variable and two vertices $u$ and $v$ are connected if and only if they at least have one constraint in common. The edges can then be expressed as the constraints. The second partition of the graph consist of all the invariants. The

### 5.1 Creating the Update Graph

## 6 Structure (or Architecture?) of My Solver

The purpose of this solver is to combine the simplicity of constraint programming formulation with the efficiency local search. Constraint programming

**Update Graph** Finding One-way constraints

```
Q = model.getIntegerVariables()
int layer = 1
while Q! = ∅ do
  for IntegerVariable var in Q do
    for Constraint cons in var.usedInConstriant() do
      if canBeMadeOneway(var,cons,layer) then
        Q.remove(var)
        break
      end if
    end for
  end for
  layer++
end while
```

**Update Graph** canBeMadeOneway(**IntegerVariable** var, **Constraint** cons, **int** layer)

```
Q = model.getIntegerVariables()
int layer = 1
while Q! = ∅ do
  for IntegerVariable var in Q do
    for Constraint cons in var.usedInConstriant() do
      if canBeMadeOneway(var,cons,layer) then
        Q.remove(var)
        break
      end if
    end for
  end for
  layer++
end while
```

sometimes offers a natural way to write problem formulations that can lead to very short and precise formulation. Local search is effective at searching for an improvement of a current solution.
This solver is roughly split into three parts, one for formulating the problem, one for finding an initial solution and a part for improving the current solution. The user needs to define variables and constraints for the problem and give and need to specify an objective for optimization. The formulation given by the user is then parsed to a constraint programming solver Gecode and Local search framework. Gecode is used to find an initial solution that the local search framework will try to improve with regard to the objective given by the user.

## 6.1 Parsing the Problem Formulation

## 6.2 The User Surface

## 6.3 The Local Search