A General Purpose Local Search Solver

November 25, 2015

Contents

1 Introduction

1.1 Discrete Optimization

1.2 Constraint Programming

1.3 Heuristics and Local Search

1.3.1 Construction Heuristics

1.3.2 Local Search and Neighborhoods

1.3.3 Metaheuristics

2 Modeling in CBLS

(Tror ikke det er så sammenhængende, mangler en mere blød overgang.)

2.1 Variables

Models contains a set of n variables $X = \{x_1, x_2, \dots, x_n\}$. Each variable $x_i \in X$ has a *domain* $D(x_i) \in D$ where D is the cartisian produkt 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 $D_i \subseteq \mathbb{Z} \ : \forall i$ **(should i declare a set $I = \{1, 2, \dots, n\}$?)**. The value of a variable x is denoted $V(x)$ and we will denote integer variables as $y_i \in Y \subseteq X$.

2.2 Constraints

The operant of values to variables will be restricted by a set of m constraints $C = \{c_1, c_2, \dots, c_m\}$. The set of variables to which the constraint c_j **(Drop subscript j?)** applies is called its *scope* and is denoted $X(c_j) = \{x_{j,1}, x_{j,i}, \dots\}$. The size of a scope $|X(c)|$ is called the *arity* $\alpha(c)$. The constraint c_j is a subset of the cartesian product of the domains of the variables in the scope $X(c_j)$ of c_j, ie, $c_j \subseteq D(x_{i_1}) \times D(x_{i_2}) \times \cdots \times D(x_{i_{\alpha(c_j)}})$.
The Constraint Satisfaction Problem (CSP) can then be defined as a triple $\mathbb{P} = \langle X, D, C \rangle$. A *solution* to the CSP $\mathbb{P}$ is a vector of n elements $\mathcal{T} = (\tau_1, \tau_2, \dots, \tau_n)$ where $\tau_i \in D_i$. The solution is feasible if $\mathcal{T}$ can be projected **(definition needed)** onto c_j for all $c_j \in C$. **(Er det helt rigtigt?)**
The questions to a CSP could be to report all feasible solutions $sol(P)$, any

feasible solution τ $insol(\mathbb{P}$ or if there exists a solution τ or not.
The CSP $\mathbb{P}$ can be expanded to a Constraint Satisfaction Optimization Problem (CSOP) $\mathbb{P}'$ with an objective function $f(\tau)$ that evaluates the quality of the solution τ, $\mathbb{P}' = \langle X, D, C, f(\tau)\rangle$. The task is then to find a solution $\hat{\tau}$ that gives minimum or maximum value of $f(\hat{\tau})$ depending on the requirements of the problem.

2.2.1 Implicit Constraints

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

2.2.2 Invariants and One-way Constraints

Invariants are variables, whose value are functionally defined by other variables. Invariants are introduced by the solver and neither the variable defined nor the variables defining the variable need not to be decision variables in the CSP or CSOP but can be auxiliary variables whose value are of interest.
One-way constraints are constraints that defines the value of an invariant.

$$x = f(\mathbf{z}) \tag{1}$$

$f(\mathbf{z})$ is a **One-way constriant** with a set of input variables z that functionally defines the **Invariant** x.

2.2.3 Soft Constraints

Soft constraints are constraints that the CBLS should try to satisfy when making neighborhood operations. If a soft constraint is not satisfied we say it is violated and it contributes an amount of violation to the problem. The amount of violation contributed depends on the type of constraint and how much it is violated.

2.3 Objective Functions

3 Previous Work

3.1 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 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 are 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 3.1.2.

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.

3.1.1 Invariants

3.1.2 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().

3.2 Gecode

3.3 LocalSolver

3.4 OscaR

4 Preprocessing and Initial Solution

4.1 Domain Reduction

4.2 Finding an Initial Solution

5 Structuring Local Search Model

Once an initial solution to the problem has been found by Gecode the model is transformed to create a model better suited for local search. Two new structures **(concepts, objects? Not sure what to call them)** are introduced in this section, dependency directed graph **(directed dependency graph?)** in subsection 5.1 and propagation queue in subsection 5.2.
When a variable x changes value other variables dependent on the value of x will need to be updated. To update those variables and invariants a directed graph $G = (V, A)$ is made, called dependency directed graph, *DDG*. The vertices V either represent a variable or an invariant and there is a arc from vertex v to vertex u if the value of the invariant represented by u depends on the value of the variable or invariant of v.
From now on when talking about vertices in G it refers to the variable or invariant it represents unless other stated.

When one or more variables change value they propagate their change to the invariants pointed in G that might point to other invariants and propagate their changes to them. We only want to visit each vertex in the graph G at most one time when making changes to on or more variables to increase performance. For each variable used in the local search a *propagation queue* q is made. A propagation queue is the order of which invariant to update when the associated variable changes value. **(Not quite happy about the description here)**

The following changes to the model are made before the local search can begin.

- Define integer variables by oneway constraints
- Define binary variables by oneway constraints

- Create auxiliary variables for the remaining constraints
- Create a dependency directed graph
- Create propagation queue for all non-fixed and non-defined variables

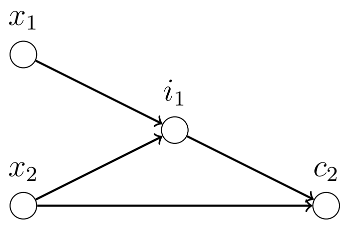
5.1 Simplification and Dependency Digraph

(What is the DDG?)
(Why do we want a DDG?)
(What properties should it have and why?)
The dependecy directed graph (DDG) G variables and invariants. The variables are non-fixed and non-defined and only have outgoing arcs. The invariants represents variables that are defined by oneway constraints or they can represent auxiliary variables used in the local search. Variables and invariants has an outgoing arc to the invariants that depends on their value.
The DDG is needed in order to evaluate and update values of variables and invariants during local search. The variables should

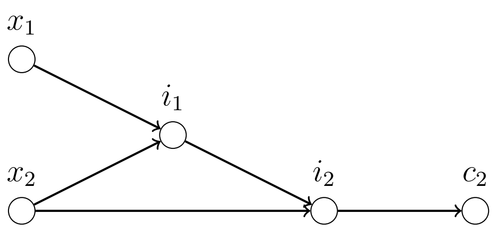
Let us consider the following example with three variables and a single constraint.

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

Initially G would consist of the three variables x_1, x_2, and x_3 and the constraints c_1 and c_2. The variable x_3 could be defined as an invariant i_1 by transforming c_1 to a oneway constraint. Once a variable is defined by a oneway constraint it is removed from the graph. Since c_1 is transformed it is removed from the G and replaced by invariant i_1 which has ingoing arcs from x_1 and x_2.



Auxiliary variable can be useful to speed up local search and in this example we could create an auxiliary variable a_1 which value is the sum of the left hand side of c_2. The auxiliary variable a_1 will be represented by an invariant i_2 which will be added to G. The invariant i_1, representing x_3, and variable x_2 have an outgoing arc to i_2 and i_2 have an outgoing arc to c_2.



(Talk about updating x_2)

(start on algorithms)

The DDG G is initially a bipartite graph where all the decision variables X are vertices in one set and all the constraints C are vertices in the other set. If constraint c applies to a variable x then there is an arc from the vertex representing x to the vertex representing c.

First all integer variables Y in the CSOP $\mathbb{P}$ **(Should I call the models CSOP or just model?)** get defined by oneway constraints by algorithm 1. If some of the integer varibles cannot be defined **(Currently report fail and exit)**.

Algorithm 1: Defining integer variables by one-way constraints

input : A set Y of integer variables sorted by decreasing domain size

output: A CBLS model for local search

```
bool $change$ = true
while $Y \neq \emptyset$ and $change$ do
    $change$ = false
    foreach $y_i \in Y$ do
        select Variable $y_i$ from $Y$
        foreach Constraint $c_j$ in $C(y_i)$ do
            if intVarCanBeMadeOneway($y_i$,$c_j$) then
                makeIntVarOneway($y_i$,$c_j$)
                Remove $y_i$ from $Y$
                isOneway($c_j$) = true
                $change$ = true
                break
            end
        end
    end
end
```

For each of the variables y_i each of the constraints c_j that applies to y_i is checked if it can be made oneway to define y_i until one is found or none can be found. This is done until there are no integer variables left or the

remaining integer variables cannot be defined, when the boolean change is false after line 15. Algorithm 2 checks if constraint c_j can be made oneway to define y_i and algorithm 3 transforms c_j into a oneway constraint defining y_i and updates the DDG G.
Let $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ be the family of objective functions.

Algorithm 2: intVarCanBeMadeOneway(Variable y_i, Constraint c_j)

```
input  : Variable $y_i$ and Constraint $c_j$
output: Boolean
1  if $c_j$ already defines a oneway constraint then
2      return false
3      (This constraint could be removed in $O(\alpha(c_j))$)
4  end
5  int $undefinedIntVar = 0$
6  foreach Variable $x$ in $c_j$ do
7      if $x$ is undefined integer variable then
8          $undefinedIntVar++$
9      end
10 end
11 if $undefinedIntVar > 1$ then
12     return false (Insures no cycle is created)
13     // else only $y_i$ is undefined integer variable
14 end
15 if $c_j$ is linear equality then
16     return true
17 end
18 if coefficient $c_{ij} \geq 0$ then
19     return false
20 end
21 foreach $f_j$ in $\mathcal{F}$ do
22     if $f_ij < 0$ then
23         return false
24         (Not sure of to define coefficients in objective
           functions. coefficient of variable i in constraint j
           could be $c_{ij}$ but $c_j$ is used for constraint and then all
           variable $y_i$ should be $y_i$)
25     end
26 end
27 return true
```

If constraint c_j already defines another variable it cannot be used to define

y_i. Line 11 makes sure that integer variable y_i is not defining another integer variable y_k. This limits the model and increases the complexity of algorithm 1 because of the addition of the initial while loop. The condition in line 11 makes sure that no strongly connected component of size 2 or greater is created in G. **(This should be described why we dont want that and what a SCC is)**. The lines 18 and 22 returns false if the coefficient of y_i in c_j is positive or one of the coefficients in the objective functions is negative. If that is not the case the constraint c_j can define variable y_i since we restrict the lower bound of y_i by the oneway constraint and want to minimize its value in the objective functions. This gives some other complications which will be described after algorithm 3.

Algorithm 3: makeIntVarOneway(Variable y_i, Constraint c_j)

```
input  : Variable $y_i$ and Constraint $c_j$
output: Updated $G$
set $Q$                                   // new coefficient set
set $U$                                       // new variable set
// Move $y_i$ to right hand side and set coefficient to 1
foreach $x_k$ in $X(c_j) \setminus y_i$ do
    $c'_{kj} = -\frac{c'_{kj}}{c_{ij}}$
    $Q = Q \cup c'_{kj}$
    $U = Q \cup x_k$
end
// Move right hand side to left hand side and update
   coefficient
double $b' = \frac{B(c_j)}{c_{ij}}$
(coefficients can now be doubles (non integer))
if $c_j$ is linear equality then
    oneway $c'$ = new oneway($U$,$Q$,$b'$)
    $G = G \cup \{c'\}$
    $G = G \setminus \{c_j\}$
end
else
    (Remember to say all constraints are either LQ or EQ)
    oneway $c'$ = new oneway($U$,$Q$,$b'$)
    Invariant $c''$ = new Max($c'$,$lowerbound(y_i)$)
    $G = G \cup c'$
    $G = G \cup c''$
    $G = G \setminus \{c_j\}$
end
```

If the constraint c_j is a linear equality constraint a new oneway constraint c' is made by isolating y_i on the right side. Then c' is added to G and c_j is removed from G. If c_j is not a linear equality constraint then it is a linear constraint with an upper bound $B(c_j)$. Since c_ij, the coefficient of y_i in c_j, is negative y_i can be isolated on the right hand side so y_i is the upper bound for the left hand side. The coefficients of y_i in the objective functions are non-negative

Once all integer variables have been defined by oneway constraints, algorithm 4 is used to make each functional constraint into a oneway constraint defining a binary variable if possible.

Let X be a set of variables and $x \in X$. The subset of constraints $C(x) \subseteq C$ is the set of constraints that applies to x.
The following algorithms describe how one-way constraints a create to define a variable x.

Algorithm 4: Defining integer variables by one-way constraints

input : A set X of variables **(Sorting order?)**
output: A model better suited for local search

```
bool $change$ = true
while $X \neq \emptyset$ and $change$ do
    $change$ = false
    foreach $x \in X$ do
        select Variable $x$ from $X$
        foreach Constraint $c$ in $C(x)$ do
            bool $flag$ = canBeMadeOneway($c$,$x$)
            if $flag$ then
                makeOneway($c$,$x$)
                Remove $x$ from $X$
                $change$ = true
                break
            end
        end
    end
end
```

The algorithm tries to create invariants that define the set of variables X by one-way constraints. It uses two other algorithms `canBeMadeOneway(`c,x`)` and `makeOneway(`c,x`)`. The first algorithm checks if the **Constraint** c can be used to define **Variable** x and the second algorithm transforms c into a one-way constraint defining x.
The complexity of algorithm **??** depends on the complexity of the two other algorithms but for simplicity let us assume they do not contribute for now. Let α_{max} be the largest arity among all constraints in C and n be the number of decision variables in the input set. The size of X has decrease by at least one each time we pass line 3 except for the first time. Hence line 3 is passed at most n times. Then the complexity of algorithm **??** is $O(\alpha_{max}n^2)$.
The coefficient of a variable x_j in constraint c_i is denoted a_{ij}. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ be the family of objective functions **(Think we should discuss this Tuesday)** and the coefficient of variable x_j in f_k be a_{kj}. **(Maybe**

call it evaluation functions. Does not make sense since a_{34} refers both to constraint and obj. func)

```
Algorithm 5: canBeMadeOneway(Constraint c, Variable x)
   input  : Constraint $c$ and Variable $x$
   output: Boolean
1  if $c$ already defines a oneway constraint then
2      return false
3      (This constraint could be removed in $O(\alpha(c))$)
4  end
5  if Number of integer variables not defined $> 1$ then
6      return false
7      (Needed in order to create the right update queue)
8  end
9  if relation($c$) is $(==)$ then
10     return true
11 end
12 (The following is not at all correct, we should discuss this
   Tuesday)
13 foreach $a$ in $A(f(x))$ do
14     if $A(c,x) \cdot a > 0$ then
15         return false
16     end
17 end
18 return true
```

(Description not finished (done at all))

Algorithm 6: makeOneway(Constraint c, Variable x)

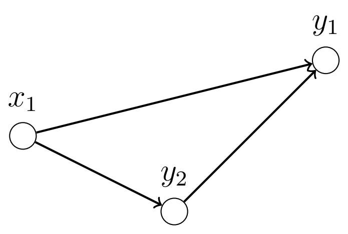
```
input  : Constraint $c$ and Variable $x$
output: An Invariant
int $coef = A_{c,x}$
$Q = A_c \backslash \{A_{c,x}\}$
$U = V(c) \backslash \{x\}$
foreach $A_{c,x}$ in $Q$ do
    $Q_{c,x} = A(c,x) \cdot \frac{-1}{coef}$
end
int $b = B(c)$
if relation($c$) is $(==)$ then
    Invariant $c' =$ Sum($U$,$Q$,$b$)
    $G = G \cup c'$
    (Maybe remove $c$ from $G$)
end
else
    Invariant $c' =$ Sum($U$,$Q$,$b$)
    Invariant $c'' =$ Max($c'$,$lb(x)$)
    $G = G \cup c'$
    $G = G \cup c''$
    (Maybe remove $c$ from $G$)
end
```

(Description of algorithm here)

Consider the following three constraints as a small example. **(Should the example be introduced before the algorithms?)**

$$\begin{aligned} c_1 : \quad & 2x_1 + y_2 - y_1 && = 2 \\ c_2 : \quad & 2x_1 - y_2 && = 2 \\ c_3 : \quad & x_1 + y_1 + y_2 && \leq 5 \end{aligned}$$

At first y_1 cannot be defined by a **One-way constraint** but y_2 can be defined by c_2 and then y_1 can be defined by c_1. The order in which the **One-way constraint** are created matters. **(Not finished here, continue tomorrow morning)**



5.2 Propagation Queue

6 Local Search Engine

6.1 Neighborhoods

6.1.1 Neighborhood Operations

7 Metaheuristics

8 Tests

9 Results

10 Conclusion

References