A General Purpose Local Search Solver

November 3, 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 $X = \{x_1, x_2, \ldots, x_n\}$. 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 $I = \{1, 2, \ldots, n\}$?)**. 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 $C = \{c_1, c_2, \ldots, c_m\}$. The set of variables to which the constraint c_j **(Drop subscript j?)** applies is called its *scope* and is denoted $X(c_j) = \{x_{j,1}, x_{j,i}, \ldots\}$. 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 $\mathcal{T} = (\tau_1, \tau_2, \ldots, \tau_n)$ 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 Structure of this CBLS (Differences from oscar and comet.)

5 Preprocessing and Simplification

5.1 Domain Reduction

5.2 Initial Solution

6 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. **(talk briefly about DDG and propagation queue (what are they used for))** procedure can be split in several steps before the local search can begin.

- Define variables by one-way constraints
- Create invariants for the remaining constraints
- Create a dependency directed graph for variables and invariants
- Create propagation queue for all non fixed variables

6.1 Simplification and Dependency Digraph

The DDG 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.
We define as many variables to be one-way constraints as possible **(This is currently not true and think this will change (one could say it is possible to define all variables and that is the optimal solution))**, starting with integer variables such that the variables in the search space in local search only consists of binary variables **(Otherwise it is not possible to solve atm.)**.

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 1: Defining integer variables by one-way constraints

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

```
1  bool change = true
2  while X ≠ ∅ and change do
3  |  change =false
4  |  foreach x ∈ X do
5  |  |  select Variable x from X
6  |  |  foreach Constraint c in C(x) do
7  |  |  |  bool flag = canBeMadeOneway(c,x)
8  |  |  |  if flag then
9  |  |  |  |  makeOneway(c,x)
10 |  |  |  |  Remove x from X
11 |  |  |  |  change = true
12 |  |  |  |  break
13 |  |  |  end
14 |  |  end
15 |  end
16 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 1 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 1 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 2: 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(α(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) · a > 0 then
15 |  |  return false
16 |  end
17 end
18 return true
```

(Description not finished (done at all))

Algorithm 3: makeOneway(Constraint c, Variable x)

input : Constraint c and Variable x
output: An Invariant

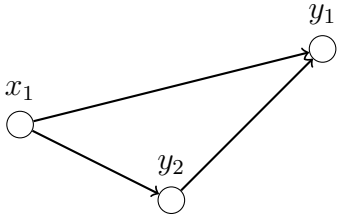
```
1  int coef = A_{c,x}
2  Q = A_c \ {A_{c,x}}
3  U = V(c) \ {x}
4  foreach A_{c,x} in Q do
5  |  Q_{c,x} = A(c,x) · (-1)/coef
6  end
7  int b = B(c)
8  if relation(c) is (==) then
9  |  Invariant c' = Sum(U,Q,b)
10 |  G = G ∪ c'
11 |  (Maybe remove c from G)
12 end
13 else
14 |  Invariant c' = Sum(U,Q,b)
15 |  Invariant c'' = Max(c',lb(x))
16 |  G = G ∪ c'
17 |  G = G ∪ c''
18 |  (Maybe remove c from G)
19 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 constriant** 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 constriant** are created matters. **(Not finished here, continue tomorrow morning)**



7 Local Search Engine

7.1 Neighborhoods

7.1.1 Neighborhood Operations

8 Metaheuristics

9 Tests

10 Results

11 Conclusion