A General Purpose Local Search Solver

November 26, 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 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 **??** 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, an invariant or a constraint.
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)**

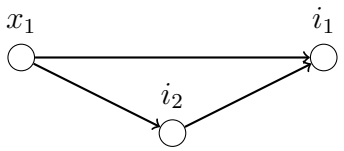
The following changes to the model are made before the local search can begin.

- 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

5.2 Propagation Queue

For each non-fixed and non-defined variable a propagation queue is made. A propagation queue q_i is an topological sort of invariants that are reachable from the vertex representing x_i in G **(maybe define topo sort)**. The propagation queue is used such that each invariant updated at most once if a variable changes value. The DDG show which invariant that are directly affected by a change in variable x_i but not the order in which they should be updated. The following example show the necessity of such an ordering.



If i_1 is updated before i_2 then it might need to be updated again after i_2 is updated hence visited twice. In worst case this could lead to a exponential number of updates instead of linear.
Propagation

6 Local Search Engine

6.1 Neighborhoods

6.1.1 Neighborhood Operations

7 Metaheuristics

8 Tests

9 Results

10 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.

11 Conclusion

References