A General Purpose Local Search Solver

December 9, 2015

Contents

List of Figures

List of Tables

1 Introduction

2 Definitions

2.1 Variables

Models contain a set of n variables $X = \{x_1, x_2, \ldots, x_n\}$ Let $I = \{1, 2, \ldots, n\}$ 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 $D = D_1 \times D_2 \times \cdots \times D_n$ 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 $D_i \subseteq \mathbb{Z} \, : \forall i$. The value of a variable x is denoted $V(x)$ and 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 $C = \{c_1, c_2, \ldots, c_m\}$. The set of variables to which the constraint c_j applies is called its *scope* and is denoted $X(c_j) = \{x_{j,i_1}, x_{j,i_2}, \ldots, x_{j,\alpha_j}\}$. The size of a scope $|X(c_j)|$ is called the *arity* α_j. 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, i.e, $c_j \subseteq D(x_{j,i_1}) \times D(x_{j,i_2}) \times \cdots \times D(x_{j,i_{\alpha_j}})$.
If all variables of a constraint c_j has a finite domain then the constraint can be written in extensional form. The extensional form of c_j is a subset of $\mathbb{Z}^{|X(c_j)|}$ of all combinations of tuples that satisfies c_j.

3 Solution Approach

3.1 Discrete Optimization

The Constraint Satisfaction Problem (CSP) is defined as a triple $\mathbb{P} = \langle X, D, C \rangle$.
A *solution* to the CSP $\mathbb{P}$ is a A vector of n elements $\tau = (V(x_1), V(x_2), \ldots, V(x_n))$.
Given a sequence $X' \subseteq X$ of variables $\tau[X']$ is called a restriction on τ with ordering 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 for each constraint c_j the restriction $\tau[X(c_j)]$ on τ satisfies constraint c_j then the solution τ is a feasible solution to the CSP $\mathbb{P}$.
The questions to a CSP could be to report all feasible solutions $sol(P)$, any feasible solution $\tau \in sol(\mathbb{P})$ 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.

3.2 Mathematical Programming

There exisist several solver for discrete optimization problems such as Cplex, Gurobi, GLPK. These solver can solver various other problems aswell. 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 can be writting on the form:

$$\text{Minimize } \mathbf{c}^T\mathbf{x} \tag{1}$$

$$\text{subject to } A\mathbf{x} \leq \mathbf{b} \tag{2}$$

$$\mathbf{x} \in \mathbb{Z}^n \tag{3}$$

Here A is a $n \times m$ matrix of coefficients, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$.
A 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.
Solving a general integer program or binary program is a NP-hard problem and several techniques are developed for solving them. The techniques can be i.e. branch and bound, cutting plane and branch and cut and usually solving an auxiliary linear problem as well. [2, p. 30]

3.3 Constraint Programming

3.3.1 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.

3.3.2 Gecode

3.4 Heuristics and Local Search

3.4.1 Construction Heuristics

3.4.2 Local Search and Neighborhoods

3.4.3 Metaheuristics

3.5 Constraint Based Local Search

3.5.1 Invariants and One-way Constraints

Invariants are variables, whose value are functionally defined by other variables and/or invariants. In this solver invariants cannot be defined by the user but are introduced by the solver, to define variables in the CSOP 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 **One-way constriant** with a set of input variables X that functionally defines the **Invariant** z.

3.5.2 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.
(Need rewrite, Van henteryck + relaxations)

3.6 Evaluation Function

3.6.1 Comet

3.6.2 OscaR

4 Architectural Overview

This section gives an overview of the key components of the solver. The following table gives an overview of the most basic classes and their ordering.

General Solver	
Gecode Engine	Local Search Engine
Model	
Constraint	Invariant
Variable	

Table 1: Ordering of key classes

The main part of the solver is the General Solverclass that functions like a distribution center. The General Solverclass contains the methods public to the user, such as creating variables and constraints, finding initial solution and optimizing the solution.

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

Local Search Engineis responsible for the optmization part of the solver with the use of local search and metaheuristics. Local Search Enginetransform the model to a CBLS model before the local search can start. How this is done and why will be discussed in section 6.

The Model class contains all variable, constraints and invariants and is used and altered by the previous three classes. Constraint and Invariant are parents to all constraint and invariant classes respectively. The contain abstract methods that the child classes must specify. The variable class is the only other class public to the user. A variable contains both the variable used by Gecode but is also used for local search.

4.1 Constraints

Constraints have some properties in common which is implemented in the parent class. All constraints have some measurement of violation. The violation can either be a zero-one measurement or it can be a measurement of how far from satisfied it is **(or how "oversatisfied" it is)**. All constraints implement must overload the methods "setDeltaViolation" and "updateViolation" from the parent class. These methods are only used during local search but are need in order to evaluate a move. **(Define Move before this?)**. The method "setDeltaViolation" calculates how much a constraint would change in violation if the move proposed is made. The method "updateViolation" is used to update the current violation of a constraint.
A user can give the constraints a priority when posting the model. This priority is used as a measurement to which of the constraints should be satisfied first.
(This section feels rather redundant since it could be done by invariants as well)

4.2 Invariants

addChange calculateDelta updateValue

5 Preprocessing and Initial Solution

5.1 Domain Reduction

5.2 Finding an Initial Solution

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
Variable bestVariable = NULL
numberOfTies = 0
foreach x_i in X(c_j) do
    if x_i is fixed or defined then
        continue
    end
    // Break ties
    if defines(x_i) < defines(bestVariable) then
        // Choose the variable that helps define fewest
            invariants
        bestVariable = x_i
        numberOfTies = 0
    end
    else if defines(x_i) == defines(bestVariable) then
        if |D(x_i)| > |D(bestVariable)| then
            // Choose the variable with largest domain
                size
            bestVariable = x_i
            numberOfTies = 0
        end
        else if |D(x_i)| == |D(bestVariable)| then
            if |deg(x_i)| < |deg(bestVariable)| then
                // Choose the variable with lowest degree
                (remember to define degree) bestVariable = x_i
                numberOfTies = 0
            end
            else if |deg(x_i)| == |deg(bestVariable)| then
                // Fair random
                numberOfTies++
                if Random(0,numberOfTies) == 0 then
                    bestVariable = x_i
                end
            end
        end
    end
end
if bestVariable ≠ NULL then
    makeOneway(Constraint c_j, Variable bestVariable)
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

set Q `// new coefficient set`
set U `// new variable set`
`// Move` x_i `to right hand side and set coefficient to 1`
foreach x_k in $X(c_j) \setminus x_i$ **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 $b' = \frac{b(c_j)}{c_{ij}}$
(Remember to define $b(c_j)$ before this) (coefficients can now be doubles (non integer))
invariant inv = new `Sum` (U,Q,b')
`// 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 2 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 7.

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.

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

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

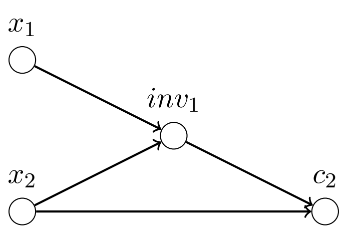

Figure 1: Small example of DDG

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 [1, 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.

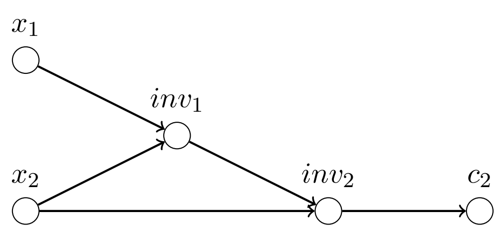

Figure 2: Small example of DDG continued

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**

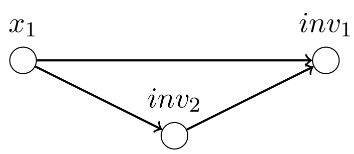

Figure 3: Importance of propagation queue

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 Local Search Engine

7.1 Neighborhoods

7.1.1 Neighborhood Operations

8 Metaheuristics

9 Tests

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)

12 Conclusion

References

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

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