A General Purpose Local Search Solver

December 2, 2015

Contents

List of Figures

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 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.

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

$f(\mathbf{z})$ 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

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 constraint satisfaction problem (CSP) has been found by Gecode the model is transformed to create a model better suited for local search, a CBLS model. 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. The dependency directed graph is used to update invariants dependent when a variable or invariant changes value. A propagation queue q_i is created for each variable x_i and it gives an ordering of the invariants directly or indirectly dependent on x_i.
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 invariants for the remaining constraints
- Create a dependency directed graph
- Create propagation queue for all non-fixed and non-defined variables

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.

5.1 Simplification and Dependency Digraph

(What is the DDG?)
(Why do we want a DDG?)
(What properties should it have and why?)
The vertices V in the dependency directed graph (DDG) $G = (V, A)$ initially only represent the non-fixed variables and the constraints. The vertex $v \in V$ has an outgoing arc to vertex $u \in V$ if and only if the value of u is directly dependent on the value of v. The variable vertices only have outgoing arcs and the constraints can only have ingoing arcs. 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, all constraint vertices to the right and the invariants vertices are added between variables and constraint vertices.
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 no longer changed directly. **(is it clear what that means? (directly))**
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 5.2.
The algorithms that creates invariants are described after this example. The example is a model with three variables and a two constraint and will illustrate how a possible dependency directed graph G is made.

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

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 can be defined as an invariant i_1 by transforming c_1 to a oneway constraint. Once variable x_3 is defined by a oneway constraint itc_1 and x_3 are removed from the graph and replaced by invariant i_1. The variables x_1 and x_2 defines i_1 hence they have outgoing arcs to i_1. Invariant i_1 has an outgoing arc to c_2 since Variable x_3 participates in c_2.

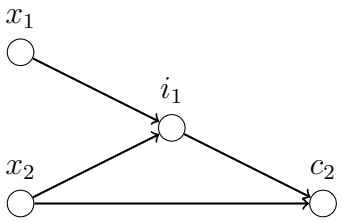


Figure 1: Small example of DDG

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 introduced 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.

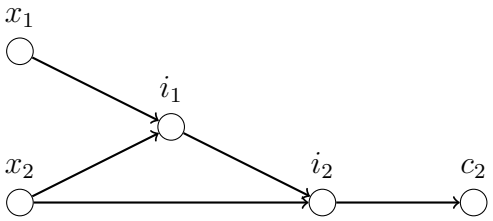


Figure 2: Small example of DDG continued

When changing the value of x_2 both invariants need to be updated since they are dependent on the value of x_2. Invariant i_2 is dependent on the value of i_1 therefore to avoid updating i_2 twice, it is beneficial to update i_1 before updating i_2. This is the ordering the propagation queue gives which is discussed in the next subsection.

For each functional constraints c_j two algorithms 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
output: Boolean

```
1  // Find the best variable to define
2  Variable bestVariable
3  numberOfTies = 0
4  foreach x_i in X(c_j) do
5      if x_i is fixed or defined then
6          continue
7      end
8      // Break ties
9      if defines(x_i) < defines(bestVariable) then
10         // Choose the variable that helps define fewest
               invariants
11         bestVariable = x_i
12         numberOfTies = 0
13         continue
14     else if defines(x_i) > defines(bestVariable) then
15         continue
16     if D(x_i) > D(bestVariable) then
17         bestVariable = x_i
18         numberOfTies = 0
19         continue
20     else if D(x_i) < D(bestVariable) then
21         continue
22     if |C(x_i)| < |C(bestVariable)| then
23         bestVariable = x_i
24         numberOfTies = 0
25         continue
26     else if |C(x_i)| > |C(bestVariable)| then
27         continue
28     // Fair random
29     numberOfTies++
30     if Random(0,numberOfTies) == 0 then
31         bestVariable = x_i
32     end
33 end
34 if bestVariable found then
35     makeOneway(Constraint c_j, Variable bestVariable)
36     return true
37 end
38 return false
```

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 which 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

1 set Q `// new coefficient set`
2 set U `// new variable set`
3 `// Move` x_i `to right hand side and set coefficient to 1`
4 **foreach** x_k in $X(c_j) \setminus x_i$ **do**
5 $\quad c'_{kj} = -\frac{c_{kj}}{c_{ij}}$
6 $\quad Q = Q \cup c'_{kj}$
7 $\quad U = Q \cup x_k$
8 **end**
9 `// Move right hand side to left hand side and update coefficient`
10 double $b' = \frac{B(c_j)}{c_{ij}}$
11 **(coefficients can now be doubles (non integer))**
12 **invariant** inv = new `Sum` (U,Q,b')
13 `// Invariant which value is (the sum of) the left hand side`
14 $G = G \cup \{inv\}$
15 $G = G \setminus \{c_j\}$
16 $G = G \setminus \{x_i\}$

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,

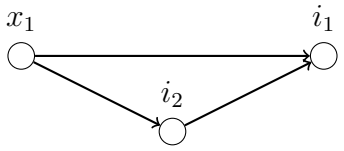
that is described in section 6.
In order to avoid circular definitions of invariants dependency directed graph G should be acyclic. A circular definition could be if x_i is used define x_j and conversely **(vise versa?)**. Then a change in value of x_i would lead to a change in value of x_j that again changes the value of x_i and so on.
Once all invariants are introduced, G is searched for strongly connected components of size two or more. 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. Should I write the algorithm down as well?)** that finds strongly connected components (*scc*) is used. Each of these strongly connected components must be removed in order to keep G acyclic, since a scc consist of at least one cycle. A scc can be removed **(made unstrongly connected?)** by removing arcs and/or removing vertices. The arcs A represents 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. In order to remove the strongly connected components one of the vertices could be removed that corresponds to undefine an invariant and reintroduce the transformed constraint and variable. Once these strongly connected components are removed there is still no guarantee that G is a directed acyclic graph (DAG). A strongly connected component can be made of several cycles and there it might not be sufficient to remove a single vertex. using Tarjans algorithm and removal of strongly connected components is repeated until no strongly connected components of size two or more is found.

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.
(Jeg føler det er meget rodet. Jeg ved ikke hvordan jeg skal dele det op)

5.2 Propagation Queue

For each non-fixed and non-defined variable x_i a propagation queue q_i is made. A propagation queue q_i is an topological sort 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 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 updated twice. In worst case updating x_i could lead to a exponential number of updates instead of linear, in the number of vertices reachable from x_i.
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 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 changes value the propagation queues are merged into a single queue removing duplicates.

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.
(Make "true" DFS when creating propagation queues, no revisit of vertices)

11 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.