

A General Purpose Local Search Solver

December 18, 2015

Contents

1	Introduction	3
2	Definitions	3
2.1	Variables	3
2.2	Constraints	3
2.3	Problem formulation	3
3	Discrete Optimization	5
3.1	Binary- and Integer Linear Programming	5
3.2	Constraint Programming	5
3.2.1	Implicit Constraints	6
3.2.2	Gecode	6
3.3	Heuristics and Local Search	7
3.3.1	Construction Heuristics	8
3.3.2	Local Search and Neighborhoods	8
3.3.3	Metaheuristics	8
3.4	Constraint Based Local Search	8
3.4.1	Invariants and One-way Constraints	8
3.4.2	Soft Constraints	8
3.5	Evaluation Function	8
3.5.1	Comet	8
3.5.2	OscAR	8
4	Architectural Overview	9
4.1	Constraints	10
4.2	Invariants	10
5	Preprocessing and Initial Solution	12
5.1	Domain Reduction	12
5.2	Finding an Initial Solution	12

6	Structuring Local Search Model	14
6.1	Simplification	14
6.2	Dependency Digraph	17
6.3	Propagation Queue	19
7	Local Search Engine	21
7.1	Neighborhoods	21
7.1.1	Neighborhood Operations	21
8	Metaheuristics	21
9	Tests	21
10	Results	21
11	Future Work	21
12	Conclusion	21

List of Figures

1	Small example of DDG	17
2	Small example of DDG continued	18
3	Importance of propagation queue	19

List of Tables

1	Ordering of key classes	9
---	-----------------------------------	---

1 Introduction

2 Definitions

2.1 Variables

Models contain a set of n variables $X = \{x_1, x_2, \dots, x_n\}$ and let $I = \{1, 2, \dots, i, \dots, n\}$ 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 $D = D_1 \times D_2 \times \dots \times D_n$ 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 $C = \{c_1, c_2, \dots, c_m\}$ and let $J = \{1, 2, \dots, j, \dots, m\}$. The set of variables to which the constraint c_j applies is called its *scope* and is denoted $X(c_j) = \{x_{1j}, x_{2j}, \dots, x_{\alpha_j j}\}$. The variable x_{ij} is the i 'th variable in constraint c_j and corresponds to a variable $x_k \in X$. The size of a scope $|X(c_j)|$ is called the *arity* α_j . The constraint c_j can be expressed as a subset of the Cartesian product of the domains of the variables in the scope $X(c_j)$, i.e., $c_j \subseteq D(x_{j,i_1}) \times D(x_{j,i_2}) \times \dots \times D(x_{j,i_{\alpha_j}})$.

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 . **(Same som cartesian product?)**

We call a constraint c_j a *functional constraint* if given an assignment of values to all variables except x_i in c_j , then at most one value of x_i satisfy c_j for all $x_i \in X(c_j)$. In other words the value of a variable in a functional constraint can be determined from the values of the other variables in the functional constraint. **(Equational constraint?)**

2.3 Problem formulation

A *Constraint Satisfaction Problem* (CSP) is defined as a triple $\mathbb{P} = \langle X, D, C \rangle$.

A *candidate solution* to a CSP \mathbb{P} is a vector of n elements $\tau = (V(x_1), V(x_2), \dots, V(x_n))$.

Given a sequence $X' \subseteq X$ of variables $\tau[X']$ is called a restriction on τ , ordered according to X . If the restriction $\tau[X(c_j)]$ matches a tuple of the constraint c_j in extensional form the solution τ satisfies constraint c_j . If each constraint $c_j \in C$ is satisfied then the solution τ is a *feasible solution* to the CSP \mathbb{P} .

(Har tænkt på et flytte nedenstående til en ny section (Solutions))

og introduce search space)

For a CSP the questions of interest 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 Optimization Problem* (COP) \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 feasible solution $\hat{\tau}$ that gives minimum or maximum value of $f(\hat{\tau})$ depending on the requirements of the problem.

3 Discrete Optimization

3.1 Binary- and Integer Linear Programming

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 (ILP) can be written in the form:

$$\text{Minimize } z = \mathbf{c}^T \mathbf{x} \quad (1)$$

$$\text{subject to } A\mathbf{x} \leq \mathbf{b} \quad (2)$$

$$\mathbf{x} \in \mathbb{Z}^n \quad (3)$$

Here A is a $n \times m$ matrix of coefficients, $\mathbf{b} \in \mathbb{R}^m$, z is the value of the objective function and $\mathbf{c} \in \mathbb{R}^n$. The first line is the objective function and can easily be transformed to a maximization problem by multiplying by -1 . The relation in line 2 can be a mix of $\{\leq, =, \geq\}$ but greater than or equal can be transformed to less than or equal by multiplying both side of the constraint by minus one.

A candidate 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. There can be several optimal solutions for a given model.

Solving a general integer program or binary program is a NP-hard problem [3, p.30] and several techniques are developed for solving them. The techniques can be i.e. branch and bound, cutting plane and branch and cut [3, p.31]. An integer linear program can be relaxed by relaxing the integer constraint, line 3 changed to $\mathbf{x} \in \mathbb{R}^n$, on the variables creating a linear programming problem [3, p. 30]. The linear programming problem is easier to solve than the integer programming problem and can be used to find bounds on the integer programming problem. The linear formulation can be transformed to *standard form*, all constraints are equality constraints, by introducing auxiliary variables and then solved using the simplex method. **(Jeg ved ikke om jeg skal uddybe det mere i.e Dual problem, når jeg kun beskæftiger mig med BP/IP)**

There exist several solver for (integer) linear programming problems such as Cplex, Gurobi, GLPK and Scip.

3.2 Constraint Programming

Constraint programming involves defining variables and constraints like ILP but often a wide range of constraints can be used. Constraint programming is declarative programming that is the program describe the desired result and not the commands or steps to reach it, just like ILP. Constraint programming uses some interface either a programming language or a frame-

work that have procedures implement to solve the problem according to the constraints posted. The language or framework may provides global constraint that can be used to formulate the problem. An example of a global constraint is the `alldifferent(x)` constraint that specify the variables `x` must have pairwise distinct values.

Two important aspects of solving a CSP are inference and search. *Inference* is adding constraints to the CSP that does not eliminate any feasible solution but might make it easier to solve the CSP [1, p.301]. Interference when dealing with variables with finite domain can local constraint propagation that is often used to eliminate large subspaces of the search space ([Search space not defined yet](#)). Propagation can be restricting domains of variables, called filtering, or combinations of values to variabels, based on the constraint doing propagation [1, p. 169]. Propagation can be done when a constraint is created by eliminating possible assignments of the variables. I.e. by doing propagation on the constraint $x_1 + x_2 \leq 2$ where $x_1, x_2 \in \mathbf{Z}^+$ we can reduce the domain of the variables to $\{0, 1, 2\}$. If we set $x_1 = 1$ we can again do propagation and reduce the domain of x_2 to $\{0, 1\}$.

([Tror jeg vil forklare om search space tidligere og introducere hvordan det bliver søgt her](#))

Search strategies explores possible assignments of variables and an exhaustive search would be a combination of all possible assignments of values to the variables. When combining propagation and search strategies the search space can be examined exhaustively and large subspaces can be pruned by propagation.

([Rettet hertil](#))

3.2.1 Implicit Constraints

Implicit constraints 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.

3.2.2 Gecode

Gecode (generic constraint development environment) is a constraint programming solver implemented in C++ and offer a wide range of modeling features. Gecode offers more than 70 constraints from the “Global Constraint Catalog” [4] that can be applied to boolean, integer, set and float variables. ([Muligvis Gecode architecture billed](#))

A model created for Gecode is created by inheriting the space class. space is is a basic layer in Gecode that a user can build the model on. To Create variables or post constraints the user need to specify the space they should be created in. When variables are created in a Gecode space views are created accordingly. Views are not used in modeling but are used to know when

propagation should be made on a constraint. When posting constraints in a space, Gecode creates propagators and these propagators can subscribe to the views of the variables. When variables changes domain the corresponding view tell its subscribes that the variables domain has changed. For some constraint the user has the option to choose the propagator based on consistency level choice. The cost of different consistency level varies from linear in the number of variables to exponential [5, p.57].

Once a problem has been formulated, the user must define on which variables and how branching is done. Just like variables and constraints are posted in a space the branch order is also posted on the space. The choice in branching involves choosing a set of variables, which of those variables to branch on first and what values to branch on. One can post several branch methods and they are treated in the order they are posted. If no branch strategy is chosen for a variable then branching is not done on that variable. To start the search a search engine must be chosen and Gecode offers two, a depth first search engine and a Branch and Bound engine. Search engines have an option class in which several options can be set [5, p.157].

When searching for a solution in a space, the search can be illustrated as a binary tree where the edges are branch choices for a variable and the vertices are the space created because of those choices. If it reaches a point where no solution is possible it stop branches from that vertex and the space is said to be failed. While searching for solution sometimes, based on a search parameter from the option given, Gecode clones the spaces. When Gecode reaches a failed space, instead of starting from scratch and recompute all way to down to the previous vertex, it uses the closest clone to backtrack to that space.

3.3 Heuristics and Local Search

In contrast to integer programming and constraint programming, local search does not do an exhaustive search for a solution hence it cannot guaranty to find an optimal solution. Where as integer programming and constraint programming does a systematic search to eliminate solution candidates, local search is based on looking at close area of the search space and look for improvements there. Local search algorithms sacrifices the optimality guaranty for performance. On many problems local search finds high quality solution within a relative search time limit.

NP-hard problems are very hard to solve optimally if the search space gets large. Integer programming usually works on the problem in different ways and can sometimes give bounds on the problem solution. Constraint programming works by eliminating some candidate solution and reducing domain of variables. Local search finds a solution and keeps searching close by for a better solution.

Construction heuristics

3.3.1 Construction Heuristics

3.3.2 Local Search and Neighborhoods

3.3.3 Metaheuristics

3.4 Constraint Based Local Search

3.4.1 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 constraint** with a set of input variables X that functionally defines the **Invariant** z .

3.4.2 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 henterick + relaxations)

3.5 Evaluation Function

3.5.1 Comet

3.5.2 OsaR

4 Architectural Overview

This section gives an overview of the key components of the solver. The table 1 gives an overview of the most basic classes and their ordering.

The main part of the solver is the General Solver class that functions like a distribution center, distributing tasks given by the user to the right part of the solver. The General Solver class 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 Engine and Local Search Engine that find the initial solution and optimize the solution respectively. Gecode Engine is 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 Engine is responsible for the optimization part of the solver with the use of local search and metaheuristics. Local Search Engine transform 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.

General Solver	
Gecode Engine	Local Search Engine
Model	
Constraint	Invariant
Variable	

Table 1: Ordering of key classes

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)

Constraints are all derived from the same class that force some method and parameters to be implemented. All constraints needs a priority according to how important the constraint is. The priority do not need to be different for the constraints but it will help the local search to differentaite between solution. (**which? Maybe a UML like box that shows the method and most important parameters**)

A constraint is posted in the constraint programming environment and later handled by local search environment. The constraints are treated differently in the environments and need different parameters and methods for that. For the CP environment few special parameters are need such as the integer consistency level (ICL) (**What more?**). The LS environment handles constraints through invariants hence a constraint needs a method for creating the invariant(s) needed in LS for the specific type of constraint.

(Example)

The Linear constraint has by definition variables, coefficients, a relation and a right hand side. When posting the constraint in Gecodes environment an ICL argument can be useful for guiding Gecode. In the LS environment the constraint is handled by creating two invariants, one for the value of the left hand side and one to determine if the constraint is violated. Implementation of invariants is described in next subsection and how the invariants are created is described in section 6.

4.2 Invariants

Invariants are all derived from one common class just like constraints. Invariants are only introduced after an initial solution to the problem is found and

before the local search has begun. Invariants can represent either variables or auxiliary variables and are defined by oneway constraints. All Invariants have the methods `addChange` and `calculateDelta` that are used during local search. The `addChange` is used to tell an invariant that a variable it depends on has been changed. The `calculateDelta` is used to update the invariant according to the changes recieved.

Each type of invariant must implement its own method since the methods can be different for each type of invariant.

Some of the classes created in the Local Search Engine uses invariants but do not differentiate between them. If invariants did not have a common parent class then each invariant type would need its own data structure for storage. Another benefit is the search procedures do not have to exam which invariants the model consist of since they all have the same methods. It also makes it easier to add new invariants since all the functionality is implemented by the new invariant and nothing has to be changed in the Local Search Engine.

5 Preprocessing and Initial Solution

Each time a constraint is posted by a user it is parsed from General Solver to the Gecode Solver. The Gecode solver is derived from Gecode Space (**Define structure / Space before this**) and the constraints parsed is posted in the Gecode Solver. Gecode has a large selection of constraints that can be used [5, p. 58-80]. Most of the constructors of these constraints are overloaded such that they can take different arguments and works with different types of variables.

5.1 Domain Reduction

Every time a constraint is posted in the Gecode Solver, Gecode does cheap propagation to reduce the domain of the variables in the constraint posted [5, p. 19].

When the user request an initial solution to the model, Gecode **Branch** method is called that specify which variables to branch on and what values (**Default or by the user?**). Then Gecode's **status** is called that does further propagation on the model before searching for a solution. If the domain of a variable is reduced to a single value, the variable is said to be fixed and is assigned that value.

5.2 Finding an Initial Solution

Once domain reduction preprocessing has been done a Gecode DFS search engine is started. The user can by an option class specify when Gecode search stop searching for a solution. A Gecode search engine takes a space and search option as arguments and the search option contains a stop object. The stop object can either be **timestop**, **nodestop** or **failstop**. Each time Gecode branches on a variable two new nodes is created and **nodestop** set a upper bound on the number of nodes to explorer. If Gecode reaches a node that gives an infeasible assignment to a variable then that space is failed and **failstop** set an upper bound on the number of fails that can occur before stopping. **Timestop** stops the search if the **timelimit** is reach.

Instead of having only one of these stop object a **multistop** object has been created that combines all three stop objects if the user wants to having multiple stop criteria.

The two following questions will be answered in this subsection. Why stop the search for a solution? What happens if the search is stopped?

Combinatorial problems can be formulated with Gecode and these problems can be very difficult to solve. In these cases Gecode keeps searching for a solution until it finds one (or runs out of available memory). Instead the search can be stopped using stop object and the problem can be relaxed such that Gecode can find a feasible solution to the relaxed problem. If one

of the stop criteria is reached we relax the problem by removing some of the constraint. To remove constraints we need to create a new Gecode Solver and repost all the constraints, except those remove, again. In order to choose which constraint that should be removed the priority of the constraints, given by the user, is used. Different techniques can be applied for choosing which constraints with the same priority to remove. The one chosen here is a stepwise backward algorithm, that is a greedy approach. **(Article Marco talked about propose some method, but says a greedy approach is good as well?)**. The number of constraints that are relaxed/removed is based on the number of constraints $|C|$ and the number of restarts made r . The number of relaxed constraints is given by equation 5.

$$|Relaxed| = \left\lceil \frac{2^{r-1}}{100} \cdot |C| \right\rceil \quad (5)$$

After seven relaxations 64 % of the constraints have been relaxed and if no solution can be found within the search limits, we report the problem is to difficult to solve within the search limit.

The constraints are chosen by priority and ties are broken at random. The constraints removed are not removed entirely from the model, they are still applied when doing local search **(Keep them with same priority as the others?)**.

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  $c_j$ 

1  Variable  $bestVariable = \text{NULL}$ 
2  numberOfTies = 0
3  foreach  $x_i$  in  $X(c_j)$  do
4      if  $x_i$  is fixed or defined then
5          | continue
6      end
7      // Break ties
8      if  $\text{defines}(x_i) < \text{defines}(bestVariable)$  then
9          | // Choose the variable that helps define fewest
          |   invariants
10         |  $bestVariable = x_i$ 
11         |  $numberOfTies = 0$ 
12     end
13     else if  $\text{defines}(x_i) == \text{defines}(bestVariable)$  then
14         | if  $|D(x_i)| > |D(bestVariable)|$  then
15             | // Choose the variable with largest domain
             |   size
16             |  $bestVariable = x_i$ 
17             |  $numberOfTies = 0$ 
18         end
19         else if  $|D(x_i)| == |D(bestVariable)|$  then
20             | if  $|deg(x_i)| < |deg(bestVariable)|$  then
21                 | // Choose the variable with lowest degree
22                 | (remember to define degree)  $bestVariable = x_i$ 
23                 |  $numberOfTies = 0$ 
24             end
25             else if  $|deg(x_i)| == |deg(bestVariable)|$  then
26                 | // Fair random
27                 |  $numberOfTies++$ 
28                 | if  $\text{Random}(0, numberOfTies) == 0$  then
29                     |  $bestVariable = x_i$ 
30                 end
31             end
32         end
33     end
34 end
35 if  $bestVariable \neq \text{NULL}$  then
36     | makeOneway(Constraint  $c_j$ , Variable  $bestVariable$ )
37 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  $G$ 

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    $c'_{kj} = -\frac{c_{kj}}{c_{ij}}$ 
6    $Q = Q \cup c'_{kj}$ 
7    $U = U \cup x_k$ 
8 end
9 // Move right hand side to left hand side and update
  coefficient
10 double  $b' = \frac{b(c_j)}{c_{ij}}$ 
11 (Remember to define  $b(c_j)$  before this) (coefficients can
   now be doubles (non integer))
12 invariant  $inv = \text{new Sum}(U, Q, b')$ 
13 // 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.

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

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

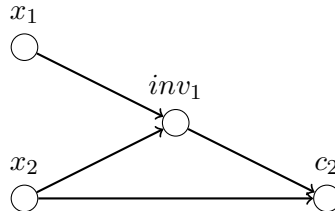


Figure 1: Small example of DDG

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 vice 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 [2, 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. Undefined 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.

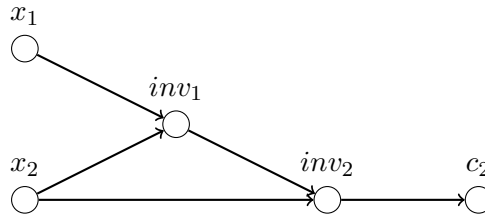


Figure 2: Small example of DDG continued

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**

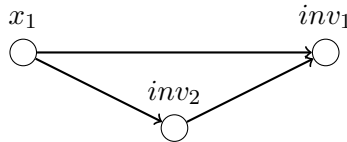


Figure 3: Importance of propagation queue

pointed to again. 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)

(option class for user to specify stop limits, branch other things)

12 Conclusion

References

- [1] Handbook of constraint programming. Foundations of Artificial Intelligence, chapter 10, pages 266–290. Elsevier Science, 2006.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [3] Jiri Matousek and Bernd Gärtner. *Understanding and Using Linear Programming (Universitext)*. Springer, 2006.

- [4] Sophie Demassey Nicolas Beldiceanu.
- [5] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling part. In *Modeling and Programming with Gecode*. 2015. Corresponds to Gecode 4.4.0.