UPPSALA
UNIVERSITET

IT 14 066
Examensarbete 15 hp
Oktober 2014

The First Constraint-Based Local Search Backend for MiniZinc

Gustav Björdal

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The First Constraint-Based Local Search Backend for MiniZinc

Gustav Björdal

MiniZinc is a modelling language used to model combinatorial optimisation and satisfaction problems, which can then be solved in a backend solver. There are many different backend solvers based on different technologies such as constraint programming, mathematical programming, or Boolean satisfiability solving. However, there is currently no constraint-based local search (CBLS) backend. This thesis gives an overview of the design of the first CBLS backend for MiniZinc. Experimental results show that for some relevant MiniZinc models, the CBLS backend is able to give high-quality results.

Handledare: Jean-Noël Monette
Ämnesgranskare: Pierre Flener
Examinator: Olle Gällmo
IT 14 066
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to thank my supervisor Dr. Jean-Noël Monette for the fruitful discussions throughout this project and for all of your help and feedback.

I would also like to thank my reviewer Professor Pierre Flener for initially introducing me to the world of constraint programming and for being the most thorough proofreader I have ever met, all of your feedback has been greatly appreciated.

Finally, I would like to thank the ASTRA group for entrusting me with this thesis project, it has been both challenging and educational, and more importantly, fun.

This project would not have been possible if it were not for the OscaR developers that are currently providing and actively maintaining the only open-source version of a CBLS framework.

Contents

1 Introduction

Combinatorial optimisation and satisfaction problems are very computationally challenging problems to solve. At the same time, they are important and have many applications within both industry and science.

Designing efficient algorithms for combinatorial problems can be a very extensive and research-heavy task. Therefore, different types of constraint solvers have emerged. A constraint solver uses a, usually solver-specific, modelling language in which the user can give a declarative definition of a problem in terms of its variables and constraints. The solver will then solve the model using some underlying technology, such as constraint programming [1], mathematical programming [2], constraint-based local search (CBLS) [3], or SAT solving [4]. In general, no solver is better than any other solver, which means that empirical tests are required to determine which solver is most suitable for a problem.

To address this, MiniZinc [5, 6] has been created as a "universal" modelling language. The goal is to allow for a "model once, solve everywhere" approach to constraint solving, which makes it easier to test different solvers. Currently, there are many solvers, based on different technologies, that provide a MiniZinc backend. However, there is no CBLS backend. CBLS is a fairly new technology that can find solutions very fast at the cost of being unable to perform complete search.

The goal of this project is to implement the first constraint-based local search backend for MiniZinc and show that such a backend has the potential to efficiently solve relevant MiniZinc models. There are two main challenges when creating a CBLS backend. First, the backend must be able to identify structures in the model and translate them into good corresponding constructs in CBLS, making use of CBLS-specific modelling techniques. Secondly, a generic search strategy is required in order to run any model without human intervention.

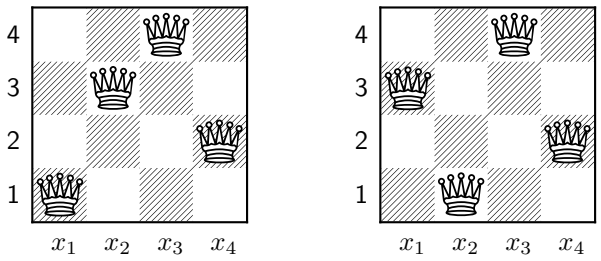
2 Background

2.1 Combinatorial Satisfaction and Optimisation Problems

A combinatorial satisfaction problem (CSP) consists of a set of variables X and a set of constraints C. Each variable $x_i \in X$ is associated with a domain $D(x_i)$ and each constraint $c_i(x_1, \ldots, x_n) \in C$ takes a number of variables as argument and is satisfied if the relationship the constraint expresses holds for the variables. A constraint is called a *global constraint* if it is parametrised in the number of arguments and expresses a common relationship that would otherwise require a conjunction of constraints or additional variables. The global constraint catalogue [7] provides an extensive overview of known global constraints. A *candidate solution* is an assignment S of every vari-

able $x_i \in X$ to a value in $D(x_i)$ and a *solution* is a candidate solution that satisfies every constraint in C.

A classical example of a CSP is the n-queens problem [8], which consists of placing n queens on an $n \times n$ chessboard such that no two queens can attack each other. A good way to express this problem is to represent the n queens with variables $x_1, \ldots, x_n$, where, in a solution S, queen i is placed in column i and row $S(x_i)$. The constraints are that no two queens are placed on the same row or diagonal, i.e., *differentRow*$(x_1, \ldots, x_n)$, *differentUpDiagonal*$(x_1, \ldots, x_n)$, and *differentDownDiagonal*$(x_1, \ldots, x_n)$. Figure 1a shows a candidate solution to the 4-queens problem, where queens 2 and 3 are breaking the *differentUpDiagonal* constraint. Figure 1b shows a candidate solution that satisfies all the constraints, making it a solution as well.



(a) (b)

Figure 1: Two different candidate solutions for the 4-queens problem.

Combinatorial optimisation problems (COP) extend CSP by associating each candidate solution with a scalar called the *objective value*. The task is then to find a solution with the minimum or maximum cost, depending on if it is a minimisation or maximisation problem.

The rest of this thesis will mainly introduce concepts in terms of minimisation problems. This can be done without loss of generality since a CSP can be viewed as a COP with a constant cost for all candidate solutions, and since maximisation problems can be transformed into minimisation problems by minimising the negated cost.

2.2 Local Search

In order to use local search so solve a COP or CSP, we will first describe the solution space as a graph. To do this, we will let each *candidate solution* be represented by a node in the graph. Each node n is then connected to each node in NEIGHBOURHOOD(n), which is the set of similar candidate solutions that can be created by performing minor modifications to n. Finally, each

node is associated with a *cost* that is given by the function Cost(n). The cost represents how good a candidate solution is.

2.2.1 Local Search for Combinatorial Optimisation Problems

For a graph G, a local search algorithm aims to find a node that satisfies some constraints, or has the lowest cost, or both. Algorithm 1 shows how local search can be summarised, where the used functions can be defined as:

- InitialNode(G) selects a node in G either using some selection strategy or randomly.
- StoppingCondition(s, *cost*) returns true when some stopping condition is met, e.g., the maximum runtime of the algorithm is exceeded or *cost* is low enough.
- Select(N) selects a neighbour based on its cost. The act of selecting and going from a candidate solution to a neighbouring solution is called a *move*.

```
1: s ← InitialNode(G)
2: while not StoppingCondition(s, Cost(s)) do
3:     N ← Neighbourhood(s)
4:     s ← Select(N)
5: return s
```

Algorithm 1: LocalSearch(G)

The name local search comes from the fact that the algorithm only explores the graph from the local perspective of s, which means that it will only keep information about s and its neighbours in memory. Furthermore, the algorithm does not maintain any information regarding the nodes it has visited nor their cost. This is the key difference between local search and systematic search, where systematic search explores graphs from a global perspective, maintaining information such as the nodes that have been visited and in which order, usually by reorganising the graph as a tree.

This means that local search requires much less memory than systematic search, which can be a great advantage when searching through very large or even infinite graphs. On the other hand, this also means that local search has no way of knowing if it has explored the entire graph or even if a node has been explored before, which means that it can move in circles. In fact local search is very prone to getting stuck in local minima, that is areas of the graph that, from a local perspective, appear to be optimal. Furthermore, local search has no way of knowing when it has found a global minimum, since it does not know if there remain any unexplored regions in the graph.

To avoid getting stuck in local minima, local search algorithms often use a *meta-heuristic*, such as tabu search, simulated annealing, or local beam search, which will strategically perform sub-optimal moves that can take the search out of local minima.

Most meta-heuristics vary greatly in terms of how they function and how well they perform on different problems. Generally, there is no meta-heuristic that outperforms all others or even is the guaranteed optimum for certain types of problems. So when it comes to selecting a meta-heuristic that works well, it is just the matter of picking one. For this reason, only tabu search will be discussed in this thesis.

2.2.2 Tabu Search

Tabu search, first introduced by Glover [9, 10], aims to keep the local search from visiting recently visited candidate solutions, by making certain moves illegal, *tabu*, for some number of iterations. To do this, each variable is given a *tabu value* and is considered to be tabu when the value is greater than the current iteration. A move is considered tabu if it involves modifying a tabu variable.

After performing a move, the tabu value of each modified variable is updated to be $it + tenure$, where it is the current iteration. This will make the variable tabu for the next *tenure* iterations. The value of *tenure* is instance-specific and usually determined using empirical tests.

Algorithm 2 outlines how local search can be modified to use tabu search. It assumes that underlying data-structures maintain the tabu values. The introduced functions can be defined as:

- NonTabu(N, it) returns a sub-set of N where each element, called a neighbour, is not tabu at iteration it.
- ModifiedVariables(s, s') returns the set of variables that are assigned to different values in s and s', i.e., the modified variables.
- MakeTabu$(V, it, tenure)$ updates the tabu value for each variable in set V to be $it + tenure$.

```
1: s ← InitialNode(G)
2: it ← 0
3: while not StoppingCondition(s, Cost(s)) do
4:    N ← NonTabu(Neighbourhood(s), it)
5:    s′ ← Select(N)
6:    MakeTabu(ModifiedVariables(s, s′), it, tenure)
7:    s ← s′
8:    it ← it + 1
9: return s
```

Algorithm 2: Tabu-Search(G, *tenure*)

Note that the described version of tabu search is very coarse when making moves tabu, as the entire domain of a variable is made tabu instead of just its previous value. This is a basic trade-off between accuracy and memory consumption. There are many different variations of tabu search: Glover and Taillard [11] provides a good overview of most standard extensions of tabu search.

2.3 Constraint-Based Local Search

Constraint-based local search (CBLS) takes the idea of having a declarative modelling language from constraint programming (CP) and combines it with local search. The declarative modelling language allows the programmer to define a problem in terms of its variables, constraints, invariants, and objective function. It is then up to the local search to find an assignment of all variables that satisfies the constraints while minimising the objective function.

2.3.1 Model

Variables Each variable of the model is given a domain and an initial value. Note that, unlike in CP, variables are at any time assigned to a value.

Invariants An invariant is a function that takes some variables as input and, as its output, always maintains the value of a variable that functionally depends on the given ones. For example, if x is supposed to be the sum $y_1 + y_2 + \cdots + y_n$, then this is expressed by using the Sum invariant, $x \leftarrow \text{Sum}(y_1, y_2, \ldots, y_n)$. Since all variables are always assigned to a value, the output of an invariant is never ambiguous, unless there are cycles of dependency between invariants.

Constraints A constraint expresses a relationship between its variables, which are given as arguments. For example, Less(x, y) states that x must

be less than y. A constraint is either satisfied or violated depending on the current assignment of its variables. There are two different categories of constraints that are treated in different ways:

- **Implicit constraints** are constraints that are satisfied by the initial candidate solution and maintained during the local search. A good example of a constraint that can be made implicit is the AllDifferent(a) constraint, which states that the values of any two variables in the array a have to be different. In the case where the variables all share the same domain and the number of variables is equal to the size of the domain, this constraint can be satisfied by initially assigning the variables to distinct values and then maintaining it satisfied by only swapping the value of two variables to create new candidate solutions. This will make sure that the values of the variables always are a permutation of the initial assignment and thus always satisfy the constraint. Van Hentenryck and Michel [12] describe several methods for identifying and maintaining several constraints as implicit constraints.

- **Soft constraints** are constraints that do not have to be satisfied by the initial assignment nor during search. Each soft constraint will instead maintain a measurement of how violated it is, referred to as its *violation*, which is updated incrementally during search as its variables are modified. How the violation is calculated is individual for every constraint, however a violation of 0 means that the constraint is satisfied. All soft constraints are added to the model's *constraint system*, which maintains the sum of the violations of all constraints, called the *total violation*. The constraint system, as well as individual constraints, can be queried for their violation given a move. The violation is also distributed among the variables of the constraint, such that each variable can be queried for its contribution to the violation.

Objective function The objective function of a model is represented by an invariant which, like soft constraints, can be queried for how a move will change its value.

2.3.2 Local Search in CBLS

Generally, a local search algorithm spends most of its time calculating the cost of candidate solutions and querying their cost change upon moves. CBLS aims to make this faster by incrementally updating the violation and objective of the current solution as moves are performed.

Furthermore, the ability, given a move, to query the model and its variables for the violation, makes it very simple to create good heuristics for a local search algorithm, such as:

1. Select one of the most violating variables.

2. Assign it a new value from its domain that results in a candidate solution with the lowest violation.

2.3.3 OscaR/CBLS

This project uses OscaR/CBLS as its CBLS framework. The OscaR [13] project is an open-source toolkit for solving operations research problems in the programming language Scala. Besides CBLS, OscaR also provides a CP solver and a few others.

Model 1 shows an OscaR/CBLS model of the n-queens problem.

2.4 MiniZinc

There are many solvers and technologies for solving combinatorial satisfaction and optimisation problems. Generally, each solver employs its own technology-specific modelling language at a certain abstraction level. This makes it difficult to experiment with different solvers as working with a new solver requires learning a new modelling language.

In an attempt to standardise modelling languages, MiniZinc [5, 6], a solver-independent medium-level modelling language, has been created. MiniZinc supports most standard modelling constructs such as sets, arrays, user-defined constraints, and decision variables of Boolean, integer, float, and integer-set types. It also comes with a library of declarative definitions of global constraints and allows solver-specific redefinitions of global constraints. MiniZinc also supports annotations, given by the modeller, that will be passed to the solver. Each annotation is prefixed with :: and is written following a value, a variable, an array, a set, a constraint, or the solve statement. Model 2 shows an example of a MiniZinc model for the n-queens problem. Note that the integer n is left unspecified. MiniZinc allows parametrised models where data is provided by separate data-files. This allows a clean separation of generic models and instance-specific data.

2.4.1 FlatZinc

To simplify the process of implementing the MiniZinc language in a solver, MiniZinc is paired with the low-level language FlatZinc. To solve a MiniZinc model, it and its data are transformed into a FlatZinc model by a process called *flattening* [16]. The resulting FlatZinc model is then presented to a solver.

A FlatZinc model only contains constants, variables, arrays, sets, constraints, a solve criterion, and annotations. This means that each expression in the MiniZinc model is realised during flattening into, possibly new, variables and constraints. When a new variable is introduced during flattening

```
1  import oscar.cbls.modeling.Algebra._
2  import oscar.cbls.constraints.core._
3  import oscar.cbls.modeling._
4  import oscar.util._
5  import oscar.cbls.invariants.core.computation.CBLSIntVar
6
7  object NQueens extends CBLSModel with App{
8    //Model
9    val N = 20
10   val tenure = 3
11   val rand = new scala.util.Random()
12
13   // initial solution
14   val init = rand.shuffle((0 to N-1).toList).toArray
15   val queens = Array.tabulate(N)(q => CBLSIntVar(0 to N-1,init(q),"queen" + q))
16   //Post the constraints
17   add(allDifferent(Array.tabulate(N)(q => (queens(q) + q).toIntVar)))
18   add(allDifferent(Array.tabulate(N)(q => (q - queens(q)).toIntVar)))
19   //Close the model and constraint system
20   close()
21
22   //Local Search
23   var it = 0
24   val tabu = Array.fill(N)(0)
25   while(violation.value > 0){
26     selectMin(0 to N-1, 0 to N-1)(
27       (p,q) => swapVal(queens(p),queens(q)),
28       (p,q) => tabu(p) < it && tabu(q) < it && p < q)
29     match{
30       case (q1,q2) =>
31         //Swap the value of queens(q1) and queens(q2)
32         queens(q1) :=: queens(q2)
33         tabu(q1)= it + tenure
34         tabu(q2) = it + tenure
35       case _ => () //Unable to find non-tabu queens
36     }
37     it += 1
38   }
39   // Output the solution
40   println(queens.mkString(","))
41 }
```

Model 1: A model of the n-queens problem in OscaR/CBLS including a search heuristic that uses tabu search. Note that the normally required AllDifferent($queens$) constraint is maintained implicitly by the swap moves. The presented code is the n-queens model found in the OscaR/CBLS documentation [14], but stripped of most comments.

```
1  include "all_different.mzn"
2  int: n;
3  array [1..n] of var 1..n: q;
4  constraint all_different(q);
5  constraint all_different([q[i] + i | i in 1..n]);
6  constraint all_different([q[i] − i | i in 1..n]);
7  solve :: int_search(q, first_fail, indomain_min, complete) satisfy;
```

Model 2: A MiniZinc model for the n-queens problem taken from [15, Section 5.1]: q is an array from index 1 to n containing decision variables with range domains from 1 to n.

it is always paired with a constraint that functionally defines it. This means that the value of the variable can always be calculated using the constraint, given the value of its other variables. Each introduced variable, say x, is given the annotations is_defined_var and var_is_introduced, and its defining constraint is given the annotation defines_var(x).

Model 3 shows the FlatZinc model generated for Model 2 when $n = 4$. Note how the argument of the all_different constraint, on line 6 in the MiniZinc model, is flattened into the new argument of the all_different constraint, on line 11 in the FlatZinc model, by introducing new variables.

```
1  predicate all_different(array [int] of var int: x);
2  var 0..3: INT____00001 :: is_defined_var :: var_is_introduced;
3  var −1..2: INT____00002 :: is_defined_var :: var_is_introduced;
4  var −2..1: INT____00003 :: is_defined_var :: var_is_introduced;
5  var −3..0: INT____00004 :: is_defined_var :: var_is_introduced;
6  var 2..5: INT____00005 :: is_defined_var :: var_is_introduced;
7  var 3..6: INT____00006 :: is_defined_var :: var_is_introduced;
8  var 4..7: INT____00007 :: is_defined_var :: var_is_introduced;
9  var 5..8: INT____00008 :: is_defined_var :: var_is_introduced;
10 array [1..4] of var 1..4: q :: output_array([1..4]);
11 constraint all_different([INT____00001, INT____00002, INT____00003, INT____00004]);
12 constraint all_different([INT____00005, INT____00006, INT____00007, INT____00008]);
13 constraint all_different(q);
14 constraint int_lin_eq([−1, 1], [INT____00001, q[1]], 1) :: defines_var(INT____00001);
15 constraint int_lin_eq([−1, 1], [INT____00002, q[2]], 2) :: defines_var(INT____00002);
16 constraint int_lin_eq([−1, 1], [INT____00003, q[3]], 3) :: defines_var(INT____00003);
17 constraint int_lin_eq([−1, 1], [INT____00004, q[4]], 4) :: defines_var(INT____00004);
18 constraint int_lin_eq([−1, 1], [INT____00005, q[1]], −1) :: defines_var(INT____00005);
19 constraint int_lin_eq([−1, 1], [INT____00006, q[2]], −2) :: defines_var(INT____00006);
20 constraint int_lin_eq([−1, 1], [INT____00007, q[3]], −3) :: defines_var(INT____00007);
21 constraint int_lin_eq([−1, 1], [INT____00008, q[4]], −4) :: defines_var(INT____00008);
22 solve :: int_search(q, first_fail, indomain_min, complete) satisfy;
```

Model 3: The FlatZinc model generated for Model 2 for instance $n = 4$.

2.4.2 The MiniZinc Challenge

Every year since 2008 various constraint solving technologies compete in the MiniZinc challenge [17, 18]. For each challenge a collection of around 100 MiniZinc model instances is gathered and used to compare solvers and solving technologies. After each challenge, the results and the model instances are published and can in turn be used to further benchmark new solvers and technologies.

2.4.3 Existing Backends

There are many backends for MiniZinc that use different underlying technologies. Here are a few of them: Gecode [19] (constraint programming), SCIP [20] (mixed integer programming), fzn2smt [21] (SAT modulo theories), and iZplus [22] (a hybrid of constraint programming and local search).

3 Design

The goal of this project is to build a CBLS backend for MiniZinc in OscaR/-CBLS that takes a FlatZinc model as input and outputs a good solution in a reasonable amount of time. The backend will, for this project, not make use of search annotations and instead work as a blackbox that solves models autonomously.

3.1 Overview of Solution

In order to do so the backend will perform the following:

1. Parse the FlatZinc model to find all of its variables and constraints as well as the solution goal. Section 3.2
2. Create a CBLS model and CBLS variables equivalent to those found in the FlatZinc model. Section 3.3
   (a) Find all variables that can be defined by invariants. Section 3.3.1
   (b) For each defined variable, turn it into the output of the invariant that defines it. Section 3.3.2
   (c) Identify suitable implicit constraints and add constructs that maintain them. Section 3.3.2
   (d) Create a constraint system for the model and post all of the non-implicit constraints. Section 3.3.2
   (e) Determine the variables that are search variables. Section 3.3.3
3. Determine a suitable heuristic. Section 3.4

4. Determine the parameters for a suitable meta-heuristic. Section 3.5
5. Determine weights for the objective. Section 3.6
6. Perform local search on the search variables using the selected heuristic and meta-heuristic. Section 3.7
7. Output solutions as they are found. Section 3.8

Each of these steps needs to be automated and can in itself be an extensive task, in which a lot of effort can be put into different types of improvements and dealing with special cases. This is especially true for determining a suitable heuristic and meta-heuristic, which, even for a given problem, can be a big research area. For this reason the goal of this project is not to create a high-performance backend for MiniZinc but rather to show that such a CBLS backend exists and is available for extensions.

The rest of Section 3 will, for each of the steps, present the design on a mainly theoretical level followed by implementation notes where necessary.

3.2 Parsing

The parser takes a FlatZinc model and translates it into an intermediate model in Scala. The intermediate model mirrors all data available in the FlatZinc model and creates appropriate data-structures, making it easier to access and manipulate the model.

Implementation Note A parser is already present in the current OscaR distribution, as a generalised version of a parser from a previous project of creating an OscaR/CP backend for MiniZinc. It is generalised in the sense that the intermediate model is neither CP nor CBLS dependent.

The parser follows the FlatZinc syntax [23, Appendix B] and supports all standard FlatZinc predicates for `int` and `bool` variables as well as `all-different` and `set_in` (with constant sets). Only variables with range domains are supported even though FlatZinc allows set domains.

3.3 Model

Once the intermediate model is obtained, the OscaR/CBLS model can be created. This is done in several steps, each refining different aspects of the model and possibly improving them.

Initially, for each variable and constant of the intermediate model, the corresponding OscaR/CBLS variable or constant is created and added to the model. A mapping between the intermediate model variables and the corresponding variables is created as well. Next, each of the created variables is given an initial value to make up the *initial candidate solution*. This is done by assigning each variable to a random value within its domain.

However note that the initial candidate solution created at this point is not necessarily the same as the one the search procedure will start from, as it can be modified in consecutive phases in order to improve the model.

Finally all variables are added to a list of variables that will be subjected to local search, called the list of *search variables*. This list will be pruned by consecutive phases of the model creation when possible in order to reduce the number of search variables.

3.3.1 Functionally Defining Variables

The variables of the intermediate model can be put in three categories:

- **Functionally defined variables** are variables that are defined by a constraint and need thus not be search variables. At this point the only known functionally defined variables are the introduced variables, which are annotated with **var_is_introduced**.
- **Annotated search variables** are variables that appear in the search heuristics annotation of the MiniZinc model. If the model creator indicates the correct variables here, then it is possible to determine the value of all other variables given an assignment of these.
- **Free variables** are variables that the search procedure needs to find a value for, i.e., all variables that are not functionally defined. It is the number of free variables that determines the size of the search space. The annotated search variables are a sub-set of the free variables.

In order to reduce the size of the search space, an attempt is made to reduce the number of free variables, by trying to identify free variables that can be functionally defined by some constraint.

There are two things to consider when doing so. First, when a constraint functionally defines a variable, the constraint will in a subsequent phase be turned into an invariant that defines the variable. Doing so can possibly increase or decrease the size of the domain of the variable. For example, if variable c is functionally defined by the FlatZinc constraint `int_plus`(a, b, c), which states that $a + b = c$, then c will be given a new domain ranging from $\min(D(a)) + \min(D(b))$ to $\max(D(a)) + \max(D(b))$, where $\min(D(x))$ and $\max(D(x))$ is the minimum and maximum value in the domain of x. This means that additional constraints may have to be posted on the variable's domain, which in turn can increase the complexity of solving the problem.

Secondly, there cannot be any circular dependencies between functionally defined variables. This means that if variable x is defined by a constraint C_1, which in turn has a variable y as one of its arguments, then y may not be defined by a constraint C_2 if any of its arguments is, to some extent, defined by x.

Furthermore, it could be possible to choose from several constraints to define a variable, but only one can be chosen. Likewise it could be possible to choose from several variables for a constraint to define, but again only one can be chosen in the end. For example, if a model contains the constraints $\texttt{int\_plus}(a, b, c)$ and $\texttt{int\_plus}(d, e, a)$, then a can be functionally defined as either $a \leftarrow \textsc{Minus}(c, b)$ or $a \leftarrow \textsc{Plus}(d, e)$, where $\textsc{Minus}(x, y)$ and $\textsc{Plus}(x, y)$ are invariants that maintains $x - y$ and $x + y$ respectively. Any $\texttt{int\_plus}(x, y, z)$ constraint can in turn define any of its variables by $x \leftarrow \textsc{Minus}(z, y)$, $y \leftarrow \textsc{Minus}(z, x)$, or $z \leftarrow \textsc{Plus}(x, y)$.

Implementation Note A lot of effort could be put into developing a good algorithm for finding the best set of free variables and the best constraints to define them. However, due to time limitations a naïve approach has been chosen instead.

Functionally defining variables can be done with two different approaches:

1. In some specific order, for each free variable, select a constraint, if any, to define it using some selection method.

2. In some specific order, for each constraint that can functionally define one of its variables, select one of its variables using some selection method and define it.

Both approaches are equivalent in that they can achieve the same results, albeit by different methods. Approach 1 is used for no other reason than that it feels more natural to implement. The free variables, including the annotated search variables, are thus considered in order of their domain size, starting with the biggest domains. This will act as a greedy method for trying to reduce the size of the search space.

When selecting a constraint to define a variable it must be ensured that it will not create a circular dependency. This is done by doing a breadth-first search of the variables the constraint depends on. If the variable we are trying to define is found, then this constraint would result in a circular dependency and another constraint is considered instead. Whether or not the variable's domain will increase is not taken into consideration when selecting a constraint. This is because, before it is decided if a variable will be functionally defined, its domain size can be said to be unknown, as it might change. Therefore, since calculating the new domain size might depend on variables with unknown domains, the task is simply too complex.

All variables that are constrained to be constants by equality constraints are defined to be invariants outside of this process. This is to make sure that the constraint is posted as an invariant, since the process described above cannot guarantee this.

3.3.2 Posting Constraints

A constraint can be posted in three different ways: as an invariant that defines a variable, as an implicit constraint, or as a soft constraint. However, since the intermediate model contains no information as to which constraints are implicit or soft the system has to deduce this information in this phase.

- **Invariants** A constraint will be posted as an invariant if it functionally defines a variable. This will only affect the introduced variables or the free variables that were turned into functionally defined variables in the previous phase. Since all the relevant information as to which constraints should be turned into invariants is already present in the intermediate model, this can be done by posting the constraint as an invariant and then removing the defined variable from the list of search variables.

  Implementation Note When an invariant functionally defines a variable, the variable is set as the invariant's output. When doing so in OscaR/CBLS, the domain of the variable will be modified to become the output domain of the invariant. This can cause two problems. To begin with, if the domain of the variable grows, then the entire problem will be relaxed and invalid solutions may become valid. Secondly, any previously made calculations or data-structures based on the size of the variable's domain will become invalid.

  To counteract this, domain constraints are posted when the domain is increased to make sure that the problem is not relaxed and all invariants are posted before all other constraints, in a topologically sorted order based on their dependencies, such that no calculations or data-structures can become invalid.

- **Implicit constraints** Each implicit constraint needs to be satisfied in the initial candidate solution and to be maintained during search by restricting how moves are made on the constrained variables. Another way to look at it is that the implicit constraint will define the neighbourhood for the affected variables. The neighbourhood for these variables will be disjoint from the other variables' neighbourhoods and generated using different algorithms. For this reason it is good to introduce the idea of neighbourhood constructs or *neighbourhood generators* [3, page 159], which define neighbourhoods for a sub-set of the search variables.

  Transforming a constraint into an implicit constraint can then be summarised in the following steps:

  1. Set the initial values of the constrained variables such that they satisfy the constraint.

2. Create a neighbourhood generator for the constrained variables that maintains the constraint.
3. Remove the constrained variables from the list of search variables and add the neighbourhood generator to the heuristic's list of neighbourhood generators.

Implementation Note The only currently supported implicit constraint is the AllDifferent(a) constraint where all variables of array a share the same domain D. The constraint can be posted implicitly using a neighbourhood generator if the following holds:

- Each element in a either is a constant, or is functionally defined to be a constant value, or is a variable in the list of search variables.
- Each variable in a has domain D and each constant value lies within D.

If this holds, then the variables in a are removed from the list of search variables and an `AllDifferentEqDom` neighbourhood generator is created.

- **Soft constraints** All constraints that cannot be transformed into implicit constraints are assumed to be soft constraints. There is no improvement to do for these constraints nor for the variables they affect, so they are just posted in the order they appear in the intermediate model.

3.3.3 Search Variables

The search variables that remain are put into a `MaxViolating` neighbourhood generator, which creates neighbours by changing the value of one variable in the current candidate solution. The neighbourhood generator is then added to the list of neighbourhood generators. In an effort to improve performance, all Boolean variables are *also* put into a `MaxViolatingSwap` neighbourhood generator, which creates neighbours by swapping the value of two variables. The justification for also using `MaxViolatingSwap` is that whenever there are two incorrectly assigned Boolean variables of opposite value their assignment can be corrected in one move instead of at least two. Boolean variables can safely be put into a swap-neighbourhood, without doing any extra calculations, since they all share the same domain.

3.4 Heuristic

A fairly simple version of *greedy hill climb* is used where each neighbourhood generator is queried for its minimum objective from which the best neighbourhood is selected and its best move is performed.

Neighbourhood generators are equipped with two different types of queries. The first is `getMinObjective`, which returns the minimum objective from a small neighbourhood that is ideally linear in size to the number of variables or their domain.

The second is `getExtendedMinObjective`, which returns the minimum objective from a neighbourhood that is a super-set of `getMinObjective`'s neighbourhood. The main reason to have two different queries is that for some models it is necessary to use the extended version in order to solve the problem but we want to use `getMinObjective` as much as possible since it is faster. Because there is no way of knowing which models require the extended version, `getExtendedMinObjective` is primarily used but both are provided such that the search procedure can switch when possible.

Implementation Note The neighbourhood generators are implemented as follows:

MaxViolating

`getMinObjective`
: Returns the lowest objective that can be achieved when assigning a highest violating non-tabu variable to another value within its domain.

`getMinExtendedObjective`
: Returns the lowest objective that can be achieved when assigning some non-tabu variable to another value within its domain.

MaxViolatingSwap

All variables of this neighbourhood generator must share the same domain.

`getMinObjective`
: Returns the lowest objective that can be achieved when swapping the value of a highest violating non-tabu variable with the value of another variable.

`getMinExtendedObjective`
: Since this neighbourhood generator is mainly used as a novelty for reassigning Booleans faster, this method is not implemented and simple calls `getMinObjective`.

AllDifferentEqDom

To initially satisfy an AllDifferent constraint, the neighbourhood generator will create a set, S, containing all values within the variables' domain D. The value of all non-variables will then be removed from S and finally, for each variable, a random value will be removed from S and assigned to the variable. Note that when the domain is larger than the number of variables, there will still be values left in S.

`getMinObjective`
: Returns the lowest objective that can be achieved when swapping the value of a highest violating non-tabu variable with the value of another variable or with a value in S.

`getMinExtendedObjective`
: Returns the lowest objective that can be achieved when swapping the value of a non-tabu variable with the value of another variable or with a value in S.

When a neighbourhood generator is queried for its best move, the neighbourhood generator will save the best move without performing it. By doing so the heuristic can sequentially query each neighbourhood generator for its best move before selecting a generator with the lowest objective. The heuristic will then tell the selected generator to commit to its best move, at which point the move is actually performed.

3.5 Meta-Heuristic

Initially all neighbourhood generators are queried for their search variables from which the set of all search variables is constructed. A mapping is then made such that each variable is given an integer value, representing its *tabu value*.

When a move is made by a neighbourhood generator it will return the variables that were affected by the move. The tabu value for these variables is then updated based on `tenure`.

When a neighbourhood generator is queried for its best neighbour, it is given a list of all non-tabu variables. The general idea is that the neighbourhood generators should only consider moves that modify non-tabu variables. However for different neighbourhood generators `tenure`, and thus the number of non-tabu variables, may be too restrictive. For this reason it is not enforced that only non-tabu variables can be modified, it is instead left up to each neighbourhood generator to respect the list of non-tabu variables.

When using tabu search, the tenure for a given instance of a problem is usually determined by either empirical tests or by some deeper understanding of the model search space. Since neither of these can be done when running the MiniZinc model on-line, where the problem is unknown beforehand, a dynamic tenure is used instead. The core concept of a dynamic tenure is that `tenure` is adjusted during search based on either the current solution, or previously visited solutions, or both.

There are several examples where some variation of tabu search with dynamic tenure has been successfully applied to problems [24, 25, 26]. However, each of these variations is both complex and based on the fact that the problem is known beforehand.

Instead, the implemented dynamic tenure is based on a *much* simpler version found in [3, page 191, Statement 10.2], which can be summarised as:

```
1: if objective < last objective and tenure ≥ MinTenure then
2:     decrease tenure by 1
3: else if tenure ≤ MaxTenure then
4:     increase tenure by 1
```

The idea is that if the objective value is improving, then `tenure` is lowered so that the current region of the search space can be explored. If the objective value is not improving, then `tenure` is increased to escape the current region.

However, this is again very problem specific, as adjusting `tenure` by 1 *every* iteration might be too extreme for some problems. Note also that the problem has shifted from finding a good value of `tenure` to finding good values for `MinTenure` and `MaxTenure`.

So, in hope of creating a more general version, the dynamic tenure is extended to keep track of the local and global minimum objective value found so far, where the global minimum is restricted to solutions with a violation of 0. Each iteration `tenure` is adjusted with the following rules:

- If enough iterations have passed without a new local minimum being found, then `tenure` is increased by `tenureIncrement` in hope of escaping the current local minimum.
- If `tenure` is greater than `MaxTenure`, then the dynamic tenure is reset by setting `tenure` to `MinTenure`, discarding the best known local minimum and setting it to the current objective. A waiting period is then initiated under which the dynamic tenure is disabled in order for the current iteration to catch up with the tabu values. The waiting period also allows the search to quickly converge into a local minimum.
- If a new best local minimum is found, then the best local minimum is updated and `tenure` is decreased by 1. The tenure is decreased in the hope that an even better minimum can be found close-by.
- If a new global minimum is found and the violation is 0, then the best local and global minimum is updated and `tenure` is set to $\max(\texttt{MinTenure}, \texttt{tenure}/2)$ and the current solution is output.

It needs to be emphasised that this design does not have any theoretical basis nor is it based on any empirical tests. Therefore no claims can be made as to how good it is for *every* problem or compared to any other design. This is however acceptable since finding such a design or even showing that it exists does not fit within the time-frame of this project, and is most likely only possible if $P = NP$.

Implementation Notes The tabu search is given the following parameters:

`MaxTenure` $= \#searchVariables \cdot 0.6$
: The value 0.6 means that at most 60% of the search variables can be tabu at the same time. The value itself has no good justification other than that it was found to work well during testing.

`MinTenure` $= 2$
: Due to the order of operations, a `tenure` of 1 means that a variable will be tabu until the *start* of the next iteration, i.e, `tenure` has no effect. A `tenure` of 2 is thus the smallest value that will have an effect on the search.

`tenure` $=$ `MinTenure`
: The initial value for `tenure` is `MinTenure` so that the search will quickly converge into a local minimum.

`tenureIncrement` $= \max(1, (\texttt{MaxTenure} - \texttt{MinTenure})/10)$
: The `tenureIncrement` causes 10% of the possible tenure values to be used at a time. The value 10 was found to be suitable during testing.

`baseSearchSize` $= 100$
: Used, after `tenure` is updated, as the minimum number of iterations that must pass before increasing `tenure`. The value 100 was found to be suitable during testing.

`searchFactor` $= 20$
: As `tenure` increases, this value determines how many extra iterations must pass before increasing `tenure` again. The value 20 was found to be suitable during testing.

When the tabu value of a variable is updated, it is assigned to:

$$\texttt{currentIt} + \min(\texttt{tenure} + \textsc{Random}(0, \texttt{tenureIncrement}), \texttt{MaxTenure})$$

where `currentIt` is the number of the current iteration.
By including RANDOM(0,`tenureIncrement`), `tenure` can be said to cover 10% of the range of possible tenure values instead of being a fixed number.

The tenure is then updated at each iteration as follows:

```
1: if the objective is a new local minimum then
2:   tenure ← max(MinTenure, tenure − 1)
3:   if the objective is a new global minimum then
4:     tenure ← max(MinTenure, tenure/2)
5: if itSinceBest > baseSearchSize + tenure + searchFactor ·
   (tenure/tenureIncrement) then
6:   tenure ← min(MaxTenure, tenure + tenureIncrement)
7:   if tenure = MaxTenure then
8:     tenure ← MinTenure
```

where `itSinceBest` is the number of iterations since a new local minimum was found.

The most questionable part here is the number of iterations that have to pass, without finding a new local minimum, before increasing `tenure`:

$$\texttt{baseSearchSize} + \texttt{tenure} + \texttt{searchFactor} \cdot (\texttt{tenure}/\texttt{tenureIncrement})$$

This expression is best understood by explaining what each part is trying to contribute:

`baseSearchSize`
: Regardless of `tenure`, some constant number of iterations should be spent at each `tenure` value, this is captured by `baseSearchSize`.

`tenure`
: After `tenure` iterations, at least `tenure` variables are guaranteed to be tabu, thus fully utilising `tenure`.

$\texttt{searchFactor} \cdot (\texttt{tenure}/\texttt{tenureIncrement})$
: As `tenure` increases, a smaller and smaller subset of search variables are up for consideration at each iteration. The content of the subset also changes with each iteration.

  Based on how much `tenureIncrement` the current `tenure` represents, extra iterations are given such that different subsets of non-tabu variables have a greater chance of being considered.

  Furthermore the runtime of each iteration decreases as `tenure` increases, allowing extra iterations.

3.6 Objective

For optimisation problems, the objective must be given by the weighted sum:

$$\texttt{violationWeight} \cdot \mathit{totalViolation} + \texttt{objectiveWeight} \cdot \mathit{objectiveVariable}$$

accompanied with a strategy for finding appropriate weights during search.

Note that *objectiveVariable* is given from the FlatZinc model and *totalViolation* is the violation of the entire constraint system. A weighting strategy is needed since *objectiveVariable* and *totalViolation* may lie within very different numerical ranges, causing one to dominate the other. Furthermore, a move may disproportionally change one of them compared to the other. For example, Model 4 shows such a case where *objectiveVariable* dominates *totalViolation*. This is because *objectiveVariable* is $1000 \cdot x$ and as x is modified, *objectiveVariable* will change by a multiple of 1000. At the same time, *totalViolation* will be given by the violation of $\textsc{Greater}(x, 90)$, for which if $x > 90$, then the violation is 0. Otherwise the violation is $1 + (90 - x)$. As x changes, *totalViolation* will thus change by a multiple of 1.

```
1  var 1..100: x;
2  constraint x > 90;
3  solve minimize 1000*x;
```

Model 4: A MiniZinc model where the corresponding *objectiveVariable* would dominate *totalViolation*, since any move will change it by a multiple of 1000 while *totalViolation* always changes by a multiple of 1. It is surprisingly hard to minimise the objective function and even satisfy this model without any weighting.

In summary two types of behaviour can be expected when the weighting is incorrect:

- If *objectiveVariable* dominates *totalViolation*, then *totalViolation* rarely reaches 0.
- If *totalViolation* dominates *objectiveVariable*, then *totalViolation* stays close to 0 but the local minimum rarely decreases.

Although these behaviours are one-way implications, a two-way implication is assumed in order to design the weighting strategy.

The weighting strategy samples *objectiveVariable* and *totalViolation* within a sample frame. If *totalViolation* did not reach 0 within the sample frame, then it is assumed that *objectiveVariable* dominates *totalViolation*, and `violationWeight` is increased in proportion to *objectiveVariable*. This is a criterion stronger than necessary and could instead be "if *totalViolation* did not decrease since the last sample frame". However by using the stronger criterion, the weighting strategy is biased towards satisfying constraints.

If on the other hand *totalViolation* does reach 0, but *objectiveVariable* does not decrease within the sample frame, then it is assumed that *totalViolation* dominates *objectiveVariable*, and `objectiveWeight` is increased in proportion to *objectiveVariable*.

Just like in the case of designing the dynamic tenure, this weighting strategy is only *good enough*, as the goal is only to have some kind of weighting strategy in place rather than a more advanced one.

For satisfaction problems, the objective is given by the *totalViolation* and no weighting occurs.

Implementation Notes The sample frame that the weighting uses is $2 \cdot$ `baseSearchSize` iterations. This is again just a value that was found suitable during testing.

At the end of each sample frame, `violationWeight` and `objectiveWeight` are changed as follows:

```
1: if violationWeight needs to be increased then
2:   if objectiveWeight > 1 then
3:     objectiveWeight ← objectiveWeight/2
4:   else
5:     inc ← max(10, |minObjectiveInSample/2|)
6:     violationWeight ← violationWeight + inc
7: if objectiveWeight needs to be increased then
8:   if violationWeight > 1 then
9:     violationWeight ← violationWeight/2
10:  else
11:    inc ← max(10, |minObjectiveInSample/2|)
12:    objectiveWeight ← objectiveWeight + inc
```

where *minObjectiveInSample* is the smallest *objectiveValue* found within the sample frame. Note that both `violationWeight` and `objectiveWeight` are integers that are never less than 1.

To make sure that the weights do not grow too large and cause integer overflows, the weights are bounded after they are changed according to:

```
1: if objectiveWeight was increased then
2:   objectiveBound ← 10000000/ max(1, abs(minObjectiveInSample))
3:   objectiveWeight ← min(objectiveWeight, objectiveBound)
4: if violationWeight was increased then
5:   violationBound ← 10000000/ max(1, minViolationInSample)
6:   violationWeight ← min(violationWeight, violationBound)
```

where `violationBound` and `objectiveBound` try to make sure that both `violationWeight` $\cdot$ *totalViolation* and `objectiveWeight` $\cdot$ *objectiveVariable* stay below 10000000, where 10000000 is chosen as an arbitrary large value.

3.7 Overall Search Strategy

When the local search is initiated it is given a maximum runtime, `MaxTime`, in milliseconds. After the maximum runtime is reached, the search is aborted.

For optimisation problems the search is performed in two phases. First it tries to only satisfy the constraints, and *objectiveVariable* is completely ignored by setting `objectiveWeight` to 0. Once the model is satisfied, `objectiveWeight` and `violationWeight` are set to 1 and the weighting strategy is activated. The main reason for doing this is that there is not really any point in trying to minimise something that we might not even be able to satisfy, therefore the initial satisfaction guarantees that the problem is even feasible.

The local search procedure also incorporates random restarts. A random restart means that every variable is reassigned to a random value within its domain that maintains any implicit constraints.

Implementation Notes Restarts are triggered differently for optimisation and satisfaction problems.

For satisfaction problems, the restart occurs after `tenure` has reached the `MaxTenure` 5 times without improving the local minimum violation, where 5 is an arbitrary cut-off at which point we believe that the next `MinTenure` to `MaxTenure` cycle will not differ from the last.

For optimisation problems, the synergy between the dynamic tenure and weighting makes consecutive `MinTenure` to `MaxTenure` cycles very different. So instead the restarts occur when `tenure` reaches `MaxTenure` and the time that has passed since the last restart or last global minimum is greater than `MaxTime`/4.

The current local minimum is discarded when a restart occurs but the global minimum is kept. This means that a run that includes a restart is not equivalent to two sequential runs with a lower `MaxTime`, since the dynamic tenure will behave differently when a low global minimum is already found.

3.8 Output

Whenever a new global minimum is found, the solution is output according to the FlatZinc format.

4 Experimental Results

4.1 Setup

In order to evaluate the translation from MiniZinc to OscaR/CBLS, the OscaR/CBLS backend is benchmarked against the model instances from the 2010 MiniZinc challenge [27].

Since the only supported global constraint is the AllDifferent, all other global constraints have to be decomposed during flattening into their MiniZinc standard library definitions [28]. Decomposing global constraints introduces a large number of variables, or constraints, or both, which makes the model more difficult to solve, or even run due to its large file-size.

Therefore, the primary reason for using the 2010 MiniZinc challenge, and not a more recent one, is that it is deemed more suitable since it only has 3, out of 11, models with unsupported global constraints.

All benchmarks for the OscaR/CBLS backend were run on a machine with the following specifications:

- Operating System: Windows 8.1
- Processor: 2.69 GHz Intel Core i7-4500U
- Memory: 8GB
- Cache: 4MB L3 cache
- Scala: Scala 2.10.4
- Java: Java 1.7

Each instance was run 10 times with a 5 minute time-out for every run, as opposed to the 15 minute time-out used in the MiniZinc challenge. The shorter time-out was chosen since running close to 100 instances 10 times for 15 minutes takes up to 250 hours. Note that since local search cannot prove optimality or unsatisfiability, instances of optimisation problems and unsatisfiable instances will always run until they time-out. The reported runtime is therefore the time it took to find the best solution or **Timeout** if no solution was found within 5 minutes.

4.2 Benchmarks

The OscaR/CBLS backend was able to run 7 of the 11 models, for which the results are shown in Tables 1–7. The column % shows the percentage of successful runs, that is runs where at least one feasible solution was found, even if suboptimal. The column **Best** contains the best objective values found by any solver in the 2010 MiniZinc challenge. The results will be discussed in Section 4.4.

The OscaR/CBLS backend was not able to run the remaining 4 models for the following reasons:

ghoulomb & rcpsp-max
: Both of the MiniZinc models for **ghoulomb** and **rcpsp-max** use the Cumulative global constraint. During flattening, the constraint decomposition causes the resulting FlatZinc files to be up to 67MB in size, which the OscaR/CBLS backend cannot handle.

wwtpp_random & wwtpp_real

Both of these problems share the same MiniZinc model, which unfortunately contains search variables with unspecified domains. When a variable has an unspecified domain it is the equivalent of its domain covering all possible values. This means that the neighbourhood generated by `MaxViolating` is very large, as there are close to 2^{32} values to consider for these variables. Therefore, not even a single iteration could be performed within the given time-out.

bacp		**Time to best solution**				**Objective value**				
Instance	%	Avg	Min	Max	Dev	Median	Min	Max	Dev	**Best**
bacp-1.fzn	100	30.95	12.98	57.58	16.10	**28**	**28**	**28**	0.00	**28**
bacp-10.fzn	100	39.49	7.94	129.13	37.98	27	27	27	0.00	**26**
bacp-11.fzn	100	19.21	3.09	54.25	13.78	**30**	**30**	**30**	0.00	**30**
bacp-12.fzn	100	51.17	10.26	235.92	66.24	**30**	**30**	**30**	0.00	**30**
bacp-14.fzn	100	36.09	20.42	61.01	13.66	29	28	29	0.50	**27**
bacp-16.fzn	100	14.99	2.69	39.20	12.72	26	**25**	26	0.40	**25**
bacp-18.fzn	100	21.94	12.56	44.79	9.28	31	31	31	0.00	**30**
bacp-2.fzn	100	71.74	7.98	295.08	86.37	**29**	**29**	**29**	0.00	**29**
bacp-21.fzn	100	7.68	2.37	15.09	3.70	27	27	27	0.00	**26**
bacp-23.fzn	100	38.81	6.94	119.50	34.22	**28**	**28**	**28**	0.00	**28**
bacp-25.fzn	100	49.54	10.12	152.45	39.66	29	29	29	0.00	**28**
bacp-27.fzn	100	33.79	17.21	91.25	20.48	**34**	**34**	35	0.49	**34**
bacp-4.fzn	100	14.51	2.20	101.41	29.09	**44**	**44**	55	4.24	**44**
bacp-6.fzn	100	62.73	29.55	150.06	33.59	**26**	**26**	39	4.22	**26**
bacp-8.fzn	100	70.70	1.83	216.37	64.54	**30**	**30**	**30**	0.00	**30**

Table 1: The time in seconds to the best solution and its objective for the **bacp** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, median, minimum, maximum, and standard deviation are presented. The column **Best** contains the best objectives found in the 2010 MiniZinc challenge. Objective values marked in bold indicate the known-to-be optimal objective values.

costas-array		**Time to satisfaction**			
Instance	%	Avg	Min	Max	Dev
14.fzn	100	10.11	0.19	21.92	6.46
15.fzn	100	39.30	7.06	88.44	25.32
16.fzn	70	88.55	23.91	197.25	67.04
17.fzn	30	112.01	62.52	149.78	36.58
19.fzn	0	**Timeout**	**Timeout**	**Timeout**	0

Table 2: The time in seconds to satisfaction for the **costas-array** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, minimum, maximum, and standard deviation are presented.

depot-placement		Time to best solution				Objective value				
Instance	%	Avg	Min	Max	Dev	Median	Min	Max	Dev	**Best**
a280_4.fzn	100	0.70	0.06	1.70	0.58	**103**	**103**	**103**	0.00	**103**
a280_6.fzn	100	4.33	0.15	10.96	3.69	**145**	**145**	**145**	0.00	**145**
att48_5.fzn	100	6.10	1.01	22.95	6.54	**13814**	**13814**	**13814**	0.00	**13814**
rat99_4.fzn	100	1.60	0.08	3.98	1.30	**105**	**105**	**105**	0.00	**105**
rat99_5.fzn	100	67.84	1.63	156.22	61.18	**107**	**107**	**107**	0.00	**107**
rat99_6.fzn	100	7.06	2.22	13.16	3.09	**114**	**114**	**114**	0.00	**114**
st70_4.fzn	100	3.13	0.79	7.00	2.07	**145**	**145**	**145**	0.00	**145**
st70_5.fzn	100	31.97	1.91	115.12	33.31	**191**	**191**	**191**	0.00	**191**
st70_6.fzn	100	97.77	2.15	293.97	99.96	**206**	**206**	210	1.96	**206**
ts225_5.fzn	100	1.00	0.72	2.08	0.37	**5000**	**5000**	**5000**	0.00	**5000**
u159_4.fzn	100	2.82	0.38	5.14	1.65	**2886**	**2886**	**2886**	0.00	**2886**
u159_6.fzn	100	86.56	14.36	150.23	54.60	5252	**5169**	5291	54.92	**5169**
ulysses16_5.fzn	100	38.21	4.06	153.44	40.85	**4331**	**4331**	**4331**	0.00	**4331**
ulysses22_4.fzn	100	3.44	1.18	6.79	1.61	**2886**	**2886**	**2886**	0.00	**2886**
ulysses22_6.fzn	100	123.22	0.95	230.74	86.50	5252	**5169**	5291	42.62	**5169**

Table 3: The time in seconds to the best solution and its objective for the **depot-placement** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, median, minimum, maximum, and standard deviation are presented. The column **Best** contains the best objectives found in the 2010 MiniZinc challenge. Objective values marked in bold indicate the known-to-be optimal objective values.

filters		Time to best solution				Objective value				
Instance	%	Avg	Min	Max	Dev	Median	Min	Max	Dev	**Best**
ar_1_1.fzn	100	138.99	83.50	268.54	49.51	36	**34**	38	1.36	**34**
ar_1_3.fzn	100	258.05	216.95	296.79	28.02	**16**	**16**	18	0.64	**16**
ar_2_4.fzn	100	282.23	261.98	299.68	10.58	18	13	27	4.35	**11**
dct_1_2.fzn	100	232.38	95.62	297.38	57.70	44	34	55	6.56	**32**
dct_2_3.fzn	100	292.84	261.50	299.97	11.25	47	38	61	6.95	**16**
dct_3_4.fzn	100	288.63	268.93	299.10	8.55	47	36	65	8.98	**11**
ewf_4_3.fzn	100	282.53	260.85	298.08	13.62	36	20	51	8.80	**17**
fir16_1_2.fzn	100	233.62	154.48	293.43	43.24	57	43	89	13.16	**19**
fir_1_2.fzn	100	160.35	101.92	251.54	49.98	**15**	**15**	**15**	0.00	**15**
fir_2_2.fzn	100	204.65	129.71	285.64	44.86	12	**11**	13	0.54	**11**

Table 4: The time in seconds to the best solution and its objective for the **filters** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, median, minimum, maximum, and standard deviation are presented. The column **Best** contains the best objectives found in the 2010 MiniZinc challenge. Objective values marked in bold indicate the known-to-be optimal objective values.

grid-colouring		Time to best solution				Objective value				
Instance	%	Avg	Min	Max	Dev	Median	Min	Max	Dev	Best
10_10.fzn	100	9.89	8.56	12.00	1.12	4	4	4	0.00	**3**
12_13.fzn	100	35.79	28.85	43.63	4.54	*4*	*4*	*4*	0.00	*4*
15_16.fzn	100	126.38	105.98	179.52	20.32	*5*	*5*	*5*	0.00	*5*
5_6.fzn	100	0.26	0.10	0.48	0.18	**3**	**3**	**3**	0.00	**3**
7_8.fzn	100	1.27	0.76	2.17	0.44	**3**	**3**	**3**	0.00	**3**

Table 5: The time in seconds to the best solution and its objective for the **grid-colouring** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, median, minimum, maximum, and standard deviation are presented. The column **Best** contains the best objectives found in the 2010 MiniZinc challenge. Objective values marked in bold indicate the known-to-be optimal objective values. Objective values marked in italics indicate the best, but not known to be optimal, objective found in the 2010 MiniZinc challenge.

solbat		Time to satisfaction			
Instance	%	Avg	Min	Max	Dev
sb_12_12_6_0.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_12_12_6_1.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_12_12_6_3.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_13_13_5_0.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_13_13_6_4.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_14_14_6_1.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_14_14_6_3.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_6_0.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_6_2.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_6_4.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_7_0.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_7_2.fzn	0	**Timeout**	**Timeout**	**Timeout**	0
sb_15_15_7_4.fzn	0	**Timeout**	**Timeout**	**Timeout**	0

Table 6: The time in seconds to satisfaction for the **solbat** problem over 10 runs. The column % shows the percentage of successful runs, of which there was none.

sugiyama		Time to best solution				Objective value				
Instance	%	Avg	Min	Max	Dev	Median	Min	Max	Dev	**Best**
g3_8_8_2.fzn	100	0.56	0.12	2.73	0.73	**2**	**2**	**2**	0.00	**2**
g3_8_8_4.fzn	100	78.72	0.27	243.08	77.92	**2**	**2**	**2**	0.00	**2**
g3_8_8_6.fzn	100	27.30	0.39	94.73	39.89	**2**	**2**	**2**	0.00	**2**
g4_7_7_7_3.fzn	100	79.15	0.53	229.00	65.99	**7**	**7**	**7**	0.00	**7**
g5_7_7_7_7_2.fzn	100	114.55	0.78	284.12	101.70	13	12	14	0.70	**11**

Table 7: The time in seconds to the best solution and its objective for the **sugiyama** problem over 10 runs. The column % shows the percentage of successful runs, for which the average, median, minimum, maximum, and standard deviation are presented. The column **Best** contains the best objectives found in the 2010 MiniZinc challenge. Objective values marked in bold indicate the known-to-be optimal objective values.

4.3 2010 MiniZinc Challenge Comparison

In the 2010 MiniZinc challenge each instance was run with a 15 minute time-out, using the following hardware:

- Operating System: Ubuntu 9.04
- Processor: 3.4Ghz Intel Pentium D (dual core)
- Memory: 4 GB
- Cache: 2MB L2 cache
- C compilers: gcc 3.4, 4.1, 4.2, 4.3
- Java: Java 1.6
- Shells: bash 3.2, tcsh 6.14, zsh 4.3

Therefore it is important to note that while it is possible to compare the results presented in Section 4.2 with the results from the 2010 MiniZinc challenge, it is very questionable to do so, especially because my hardware is faster than theirs.

Even if the results from the 2010 MiniZinc challenge used the same hardware and time-out as in Section 4.2, the results would still say very little about the potential of each underlying solver, as discussed by Stuckey et al. [17, Section 1.1]. Furthermore, the result from one instance of a problem does not necessarily generalise to all instances of the same problem, making any comparison and conclusion only valid for the specific instance. Therefore a comparison with the 2010 MiniZinc challenge is made only to provide a context for the results.

There is also the question of which runtime to use, for the Oscar/CBLS backend, when doing the comparison. Since proving optimality is impossible

in local search, comparing the total runtime provides very little value. It is instead more interesting and relevant to compare the time it took to find a solution, as local search can often find good solutions much faster than systematic search.

However, this is not really fair since the time to the best solution is not available for the solvers in the 2010 MiniZinc challenge, which bases the scores of optimisation problems on the time it takes to prove a solution optimal, called the total runtime.

But as it turns out, the scoring algorithm used in the 2010 MiniZinc challenge puts very little emphasis on the runtime of solvers and there is an insignificant difference in the final scoring when using either the total runtime or the time to the best solution for the OscaR/CBLS backend. Therefore only one comparison is made, comparing the total runtime for the OscaR/CBLS backend with the total runtime for the other solvers.

Table 8 shows the total summary for the scores from the 2010 MiniZinc challenge. The median objective and average time (for satisfaction problems) from the results in Section 4.2 have been injected into the challenge scoring algorithm under the solver **oscar_cbls_fair**. All optimisation problems as well as the unsatisfiable instances are given a runtime of 5 minutes. The suffix **fair** is used to clearly indicate that it is possible to do a different scoring that is more favourable to the OscaR/CBLS backend.

The models that the OscaR/CBLS backend is unable to solve are also included in the summary. No runtime or objective is provided for these models, which means that the scoring algorithm makes the default assumption that a time-out occurred after 15 minutes. Table 9 shows a summary for all models that the OscaR/CBLS backend was able to solve at lest once, note that this excludes **solbat**.

Table 10 shows the scores per model for the 6 solvable models, where at least one feasible solution was found by the OscaR/CBLS backend. Table 11 shows the scores per model for the 5 unsolvable models, that were the OscaR/CBLS backend was either not able to run the model or find a single feasible solution. Note that the scoring algorithm penalises solvers that are not able to prove optimality, which is why the score might seem very low for some problems, compared with the results in Section 4.2.

Score Summary For all 11 models		
Solver	**Time**	**Score**
TOTAL	422310	9900.0
chuffed_free	9645	3208.1
gecode_free	43997	1708.9
fzn2smt_free	16087	1665.7
jacop_fd	43913	880.6
fzntini_free	40937	798.5
oscar_cbls_fair	47049	521.5
g12_fd_free	70398	495.6
cplex_free	67165	357.2
scip_free	74119	264.0

Table 8: The total times and scores for the entire Free category in the 2010 MiniZinc challenge with the results from Section 4.2 injected into the scoring. **oscar_cbls_fair** uses a time-out of 5 minutes while the other solvers use a 15 minute time-out. Entries are sorted by score. Note that this includes scores for the 4 models that were not possible to run in the OscaR/CBLS backend.

Score summary for the 6 runnable models		
Solver	**Time**	**Score**
TOTAL	211622	5500.0
chuffed_free	3991	1565.8
fzn2smt_free	12058	732.9
jacop_fd	23347	627.2
fzntini_free	30347	549.1
gecode_free	24263	529.0
oscar_cbls_fair	15549	521.5
cplex_free	28696	357.2
g12_fd_free	37952	353.2
scip_free	35419	264.0

Table 9: The total times and scores for the entire Free category in the 2010 MiniZinc challenge on the 6 models that the OscaR/CBLS backend was able to solve at least once within 10 runs. The results from Section 4.2 for these models have been injected into the scoring. **oscar_cbls_fair** uses a time-out of 5 minutes while the other solvers use a 15 minute time-out. Entries are sorted by score.

bacp		
	Time	Score
TOTAL	38499	1500.0
chuffed_free	15	416.2
cplex_free	46	269.2
scip_free	288	179.3
fzn2smt_free	986	165.6
g12_fd_free	8784	124.1
oscar_cbls_fair	4500	113.5
fzntini_free	10982	84.9
gecode_free	5942	82.3
jacop_fd	6956	64.9

costas_array		
	Time	Score
TOTAL	24061	500.0
chuffed_free	1	320.1
gecode_free	1787	47.9
oscar_cbls_fair	549	47.3
g12_fd_free	2766	32.5
jacop_fd	2091	29.4
fzntini_free	3682	15.7
fzn2smt_free	4185	7.2
cplex_free	4500	0.0
scip_free	4500	0.0

depot_placement		
	Time	Score
TOTAL	66351	1500.0
chuffed_free	281	404.6
gecode_free	4877	264.8
fzn2smt_free	541	243.6
oscar_cbls_fair	4500	142.9
jacop_fd	9438	138.1
fzntini_free	10477	130.6
g12_fd_free	9292	127.2
scip_free	13445	48.1
cplex_free	13500	0.0

filters		
	Time	Score
TOTAL	41591	1000.0
jacop_fd	207	288.9
chuffed_free	1869	201.5
fzntini_free	2863	155.2
fzn2smt_free	3628	139.6
oscar_cbls_fair	3000	78.6
g12_fd_free	9000	61.2
cplex_free	4727	55.0
gecode_free	7297	20.1
scip_free	9000	0.0

grid_colouring		
	Time	Score
TOTAL	28157	500.0
oscar_cbls_fair	1500	103.9
fzntini_free	2254	90.7
chuffed_free	1812	89.1
fzn2smt_free	2706	64.9
gecode_free	3601	57.6
jacop_fd	3598	57.2
scip_free	3686	36.6
cplex_free	4500	0.0
g12_fd_free	4500	0.0

sugiyama		
	Time	Score
TOTAL	12963	500.0
chuffed_free	13	134.4
fzn2smt_free	12	112.0
fzntini_free	89	72.1
gecode_free	759	56.3
jacop_fd	1057	48.7
oscar_cbls_fair	1500	35.2
cplex_free	1423	33.1
g12_fd_free	3610	8.2
scip_free	4500	0.0

Table 10: The scores for the **bacp**, **costas_array**, **depot_placement**, **filters**, **grid_colouring**, and **sugiyama** models. **oscar_cbls_fair** uses a time-out of 5 minutes while the other solvers use a 15 minute time-out. Entries are sorted by score.

ghoulomb		
	Time	Score
TOTAL	43273	1000.0
gecode_free	1949	651.1
chuffed_free	4147	348.9
jacop_fd	7	0.0
fzn2smt_free	74	0.0
fzntini_free	1096	0.0
oscar_cbls_fair	9000	0.0
cplex_free	9000	0.0
g12_fd_free	9000	0.0
scip_free	9000	0.0

rcpsp_max		
	Time	Score
TOTAL	49919	900.0
chuffed_free	1264	317.3
gecode_free	6323	249.0
fzn2smt_free	3673	212.7
jacop_fd	8066	88.2
g12_fd_free	7246	32.8
fzntini_free	178	0.0
cplex_free	6969	0.0
scip_free	7200	0.0
oscar_cbls_fair	9000	0.0

solbat		
	Time	Score
TOTAL	65234	1500.0
chuffed_free	73	641.2
fzn2smt_free	246	321.7
gecode_free	6062	167.5
fzntini_free	4029	164.6
g12_fd_free	7200	109.5
jacop_fd	7124	95.5
oscar_cbls_fair	4500	0.0
cplex_free	13500	0.0
scip_free	13500	0.0

wwtpp_random		
	Time	Score
TOTAL	27834	500.0
fzn2smt_free	19	237.7
chuffed_free	82	170.8
fzntini_free	2558	46.4
gecode_free	3600	28.9
jacop_fd	3575	16.3
oscar_cbls_fair	4500	0.0
cplex_free	4500	0.0
g12_fd_free	4500	0.0
scip_free	4500	0.0

wwtpp_real		
	Time	Score
TOTAL	24428	500.0
chuffed_free	88	164.1
fzn2smt_free	17	160.8
gecode_free	1800	83.3
jacop_fd	1794	53.4
fzntini_free	2729	38.3
oscar_cbls_fair	4500	0.0
g12_fd_free	4500	0.0
cplex_free	4500	0.0
scip_free	4500	0.0

Table 11: The scores for the **rcpsp_max**, **solbat**, **wwtpp_random**, **wwtpp_real**, and **ghoulomb** models. Note that **oscar_cbls_fair** is unable to solve any instances for these problems; the posted time, except for **solbat**, is the default time of 15 minutes that is given by the scoring algorithm when a solver is unable to run a model. Entries are sorted by score.

4.4 Discussion

The OscaR/CBLS backend was able to run 7 of the 11 models. As shown in Tables 1, 3, and 7 it was able to find the optimal solution or the optimal solution plus 1, at least once for all instances of the **bacp**, **depot-placement**, and **sugiyama** problems. It is also interesting to note that for most instances the average time it took to find its best solution is less than one third of the total runtime.

The **grid-colouring** problem, Table 5, shows what are probably the most noteworthy results. Not only is the OscaR/CBLS backend able to find the optimal solution or the optimal solution plus 1 for all instances. The OscaR/CBLS backend is also, for **12_13.fzn** and **15_16.fzn**, able to consistently find the objective values 4 and 5. Only one other solver, chuffed_par in the parallel category, is able to find these objectives, but is not able to prove them to be optimal. This means that the OscaR/CBLS backend would be the only solver in the free category that is able to find these objective values. Although it is an unfair comparison, it can be noted that, as shown in Table 5, the total average *time to the best solution* for *all* instances is 174 seconds, which is considerably less than the other solvers total runtime as seen in Table 10.

The **costas-array** and **solbat** models, shown in Tables 2 and 6 respectively, are the only satisfaction problems in the 2010 MiniZinc challenge. Table 2 shows a clear increase in difficulty for each instance of the **costas-array** problem, starting with **14.fzn** being solved in 10.11 seconds on average to **19.fzn** being unsolvable before timing out. This is not surprising as the size of the problem does indeed grow with every instance as the results suggest.

On the other hand, **solbat** proved to be completely unsolvable before timing out as seen in Table 6, where every instance times out. It is difficult to know exactly why this problem is so difficult for the OscaR/CBLS backend to solve. But it can be noted that the smallest instance, **sb_12_12_5_0.fzn**, has close to 6000 introduced variables, roughly 5 times the largest number of introduced variables found in the other runnable problems. This could be an explanation.

filters is the only runnable model that uses unsupported global constraints, namely DiffN and Maximum. Despite the performed global constraint decomposition, the OscaR/CBLS backend is still able to find optimal solutions and shows a 100% success for all runs in Table 4.

What is interesting to note is that for the harder instances, even though the OscaR/CBLS backend is only able to find low-quality objectives, most other solvers found no solution at all. Consider for example **dct_2_3.fzn**, where the minimum objective found is 35. In the 2010 MiniZinc challenge, jacop_fd proved that the minimum objective is 16, while fzntini_free and fzn2smt_free found the objective 16 and 17 respectively before they timed-

out. All other solvers timed out before they could find any objective at all. So while the objective of 35 found by the OscaR/CBLS backend is very high, it is still better than finding nothing.

5 Conclusion

The main goal of this project was to show that it is possible to create a CBLS backend for MiniZinc with acceptable blackbox performance.

This thesis has shown how such a backend can be created and it has outlined some major problems that arise when transforming a MiniZinc model into a CBLS model.

The presented benchmark results in Section 4.2 show that the backend is able to solve some relevant MiniZinc models with high-quality results, but also that there are classes of models that require further work to be solved.

Finally, the comparison with the results from the 2010 MiniZinc challenge in Section 4.3 shows that, by a 2010 standard, the implemented backend can produce competitive results.

6 Future Work

The implemented backend with the presented results makes for a good starting point for creating a good CBLS backend for MiniZinc. As always, all aspects of the design can be improved and new features can be added in order to improve the OscaR/CBLS backend. However, at this point the improvements presented below are considered to be the most relevant and interesting.

Global Constraint Definitions

Every solver can supply solver-dependent definitions for global constraints to be used instead of a global constraint decomposition. Currently, only the AllDifferent global constraint is given such a definition and all other global constraints are decomposed. This is the reason why neither the **ghoulomb** nor the **rcpsp-max** models were runnable, as the decomposition resulted in massive FlatZinc models. The solution is to add OscaR/CBLS-dependent definitions for all the global constraints and the corresponding support in the backend. This is a time-consuming but not difficult task that was not done during this project as there was not enough time.

Set Domains

FlatZinc allows variables to have a set as domain, however OscaR/CBLS only supports range domains for its variables. Initial attempts to extend OscaR/-CBLS to support set domains were made during the development, but the

idea was abandoned as it proved to be too time-consuming. However, if OscaR/CBLS is to fully support the FlatZinc language, then this is something that needs to be done.

Set Variables

This project has focused on adding support for integers and Booleans due to time limitations. However, both OscaR/CBLS and FlatZinc allow the use of set variables, and an obvious extension to the current implementation would be to support set variables. This would mainly require new neighbourhood generators that compute neighbours for set variables.

MiniZinc Annotations

MiniZinc allows the use of solver-specific annotations as part of the MiniZinc model. The annotations could for example be used to control manually the parameters of the heuristic and the meta-heuristics. So while the current, fully automatic system is convenient during rapid development and for doing quick comparisons or participating in competitions, annotation support is crucial when using the backend in practice.

Initial Propagation

The OscaR/CBLS backend was unable to run the **wwtpp_random** and **wwtpp_real** models because they had variables with unbound domains. However, upon closer inspection of the models this is not entirely true. Each of the variables with an unbound domain has two corresponding LessEq constraints, with a constant as one of the arguments, which does in fact bound the domain. It appears that instead of giving the variables the correct domain in their declaration, the model creator has opted to declare the domain using constraints.

To fix the resulting unbound search spaces, the OscaR/CBLS backend could post all LessEq constraints that include a constant implicitly by reducing the domain of the constrained variable. This would not only make the domains and search space more manageable but also remove some constraints from the OscaR/CBLS model, which improves performance.

Furthermore, this idea can be generalised. The described countermeasure is the equivalent of, in CP, propagating the LessEq constraint so that it is subsumed and removing it from the model. The generalisation is thus, during the model creation, to propagate *all* constraints to domain consistency on the variable domains. Then remove the constraints that were subsumed and reassign the initial value of all affected variables to respect the new, possibly set, domain. This could reduce the size of the search space as well as the number of constraints and would make both **wwtpp_random** and **wwtpp_real** runnable.

It should be added that the OscaR project has both a CBLS and CP implementation making it an ideal candidate to test this kind of initial propagation on.

Furthermore, there is the iZplus [22] solver, which has earned 2 bronze medals in the 2012 MiniZinc challenge, 1 bronze medal in the 2013 MiniZinc challenge, and 1 gold in the 2014 MiniZinc challenge. iZplus is a CP and local search hybrid that use a similar technique to the suggested initial propagation. It uses CP to find an initial solution that satisfies all constraints and then uses local search to find similar solutions that improve the objective function. As there is currently no detailed publication of how iZplus works, it is not clear if the local search part is related to CBLS or if it is specialised for the task of finding similar solutions.

Intermediate Model Simplification

Improvements similar to the initial propagation can be made in the intermediate model by removing redundant variables. The IntEq(x, y) constraint states that $x = y$, which can be maintained implicitly already in the intermediate model by replacing all occurrences of x in the model by y or vice versa, ideally replacing the one with the largest domain. The same is true for BoolEq(a, b) and even Bool2Int(a, x), due to how Booleans are represented in OscaR/CBLS.

Note that this technique is available in the implementation of the intermediate model that is already present in OscaR and was used for a long time during the development. However, as it turns out, the data representation used in the intermediate model makes it very difficult to assure that all occurrences of a variable were replaced and all corner cases covered. Therefore, this is not used in the current implementation.

Neighbourhood Generators

The neighbourhood generators are currently underutilised in a few ways. To begin with, additional neighbourhood generators can be implemented to support various heuristics. Currently, each of the implemented neighbourhood generators is based on a maximally-violating variable selection scheme. However, this does not work very well when variables have very large domains, in which case a first-improving solution heuristic, which does not explore the entire neighbourhood, would be more suitable. By implementing a variety of neighbourhood generators the OscaR/CBLS backend can try to identify and use the most suitable neighbourhood generators while creating the model. It could also, for example, add all variables with too large a domain into a first-improving neighbourhood generator and those with a small domain into a maximally-violating neighbourhood generator.

Furthermore, when several neighbourhood generators are used at the

same time, the OscaR/CBLS backend currently queries all of them. This could be improved by using a selection strategy to query only one neighbourhood generator at each iteration, thus saving computation time.

It should be noted that in parallel but independently of this project, the OscaR team has started implementing neighbourhood generators as a built-in construct in OscaR/CBLS. Replacing our neighbourhood generators with the future native neighbourhood generators will be an important step forward.

References

[1] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[2] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley New York, 1988.

[3] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2009.

[4] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. IOS Press, 2009.

[5] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74969-1. doi: 10.1007/978-3-540-74970-7_38.

[6] NICTA Optimisation Research Group. MiniZinc. `http://www.minizinc.org/`, . [Online; accessed 23rd October 2014].

[7] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present, and future. *Constraints*, 12(1):21–62, March 2007. The catalogue is at `http://sofdem.github.io/gccat`.

[8] Bernard A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Intelligent Systems*, 5(3):16–23, 1990.

[9] Fred Glover. Tabu Search Part I. *ORSA Journal on Computing*, 1(3): 190–206, 1989. doi: 10.1287/ijoc.1.3.190.

[10] Fred Glover. Tabu Search Part II. *ORSA Journal on Computing*, 2 (1):4–32, 1990. doi: 10.1287/ijoc.2.1.4.

[11] Fred Glover and Eric Taillard. A user's guide to tabu search. *Annals of Operations Research*, 41(1):1–28, 1993. ISSN 0254-5330. doi: 10.1007/BF02078647.

[12] Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 273. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[13] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[14] OscaR Team. Getting started: The NQueen problem. Available from `https://bitbucket.org/oscarlib/oscar/wiki/StartCBLS` [Online; accessed 23rd October 2014].

[15] Kim Marriott and Peter J. Stuckey. A MiniZinc Tutorial. `http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf`. [Online; accessed 23rd October 2014].

[16] Nicholas Nethercote. Converting MiniZinc to FlatZinc. `http://www.minizinc.org/downloads/doc-1.6/mzn2fzn.pdf`. [Online; accessed 23rd October 2014].

[17] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010. ISSN 1383-7133. doi: 10.1007/s10601-010-9093-0.

[18] NICTA Optimisation Research Group. The MiniZinc Challenge. `http://www.minizinc.org/challenge.html`, . [Online; accessed 23rd October 2014].

[19] Gecode Team. `http://www.gecode.org/flatzinc.html`. [Online; accessed 23rd October 2014].

[20] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009. `http://mpc.zib.de/index.php/MPC/article/view/4`.

[21] Logic and Programming research group at UdG. fzn2smt. `http://ima.udg.edu/Recerca/GrupESLIP.html`. [Online; accessed 23rd October 2014].

[22] Toshimitsu Fujiwara. iZ based solver for MiniZinc Challenge 2014. `http://www.minizinc.org/challenge2014/description_izplus.txt`. [Online; accessed 23rd October 2014].

[23] Ralph Becket. Specification of FlatZinc. `http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf`. [Online; accessed 23rd October 2014].

[24] Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2006.05.014. Part Special Issue: New Trends in Locational Analysis.

[25] Ümit Bilge, Furkan Kıraç, Müjde Kurtulan, and Pelin Pekgün. A tabu search algorithm for parallel machine total tardiness problem. *Computers & Operations Research*, 31(3):397 – 414, 2004. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/S0305-0548(02)00198-3.

[26] Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994. doi: 10.1287/ijoc.6.2.126.

[27] NICTA Optimisation Research Group. MiniZinc Challenge 2010 - Rules. `http://www.minizinc.org/challenge2010/rules2010.html`, . [Online; accessed 23rd October 2014].

[28] NICTA Optimisation Research Group. The MiniZinc standard library. `https://github.com/MiniZinc/minizinc-stdlib`, . [Online; accessed 23rd October 2014].