TBD manuscript No.
(will be inserted by the editor)

MiniCP: A Lightweight Solver for Constraint Programming

L. Michel · P. Schaus · P. Van Hentenryck

Draft September 15, 2018

Received: date / Accepted: date

Abstract This paper introduces MiniCP, a lightweight, open-source solver for constraint programming. MiniCP is motivated by educational purposes and, in particular, the desire to provide the core implementation of a constraint-programming solver for students in computer science and industrial engineering. The design of MiniCP provides a one-to-one mapping between the theoretical and implementation concepts and its compositional abstractions favor extensibility and flexibility. MiniCP obviously does not support all available constraint-programming features and implementation techniques, but these could be implemented as future extensions or exploratory projects. MiniCP also comes with a full set of exercises, unit tests, and development projects.

1 Introduction

Motivation Constraint programming [4,10,24] originated from the logic-programming community in the mid-1980s. It is now used in numerous commercial applications, primarily in the areas of scheduling, routing, and timetabling, and is often hybridized with mathematical programming through decomposition techniques such as logical Benders decomposition and column generation. It is supported by several commercial solvers (e.g., CP Optimizer, an IBM product) and a multitude of open-source solvers. An interesting review of scheduling applications of CP Optimizer can be found in [13].

Yet, many undergraduate students in computer science and industrial engineering graduate without knowledge of constraint programming. The primary reason is simple: Constraint programming is simply not taught in most universities and there is a lack of high-quality teaching material in the community. Exacerbating this situation is the fact that most constraint-programming solvers have been built for speed and functionality, not for education purposes. As a result, they are often hard to penetrate and modify, and are not particularly well-adapted for use in the classroom. It is often the case that core concepts are hard to isolate among the wide variety of optimizations and features. This should be contrasted to the MiniSat solver for Boolean satisfiability which has largely contributed to the dissemination of (CDCL) SAT solvers. MiniSat has a neat, minimalist, and well-documented architecture that has enabled several generations of students and researchers to enter the field and make significant contributions.

This work is an attempt to bridge this gap: It introduces MiniCP, a light, open-source constraint-programming solver whose primary goal is to foster education in constraint programming. The hope is that MiniCP provides the core material to support classes in constraint programming at any institution

Laurent Michel
Computer Science & Engineering Department, University of Connecticut 371 Fairfield Road, Storrs, 06269-4155, CT USA
E-mail: ldm@engr.uconn.edu

P. Schaus
Institute of Information and Communication Technologies, Electronics and Applied Mathematics, Louvain School of Engineering, Louvain la Neuve, Belgium
E-mail: pierre.schaus@uclouvain.be

Pascal Van Hentenryck
H. Milton Steward School of Industrial and Systems Engineering (ISyE), Georgia Institute of Technology, Atlanta, GA 30332
E-mail: pvh@isye.gatech.edu

package	LOC
engine.core	827
engine.constraints	1265
examples	641
state	661
cp	229
search	184
util	194
unit tests	3822

Table 1: MiniCP packages and their lines of codes (LOC) (computed with `sloccount`).

with minimal effort. Not only does the MiniCP project contribute a minimalist and neat constraint-programming kernel; It also provides exercises, unit tests, and development projects. Ultimately, this educational solver will be accompanied by slides and video lectures that could be used for flipped offerings.

The Design and Implementation of MiniCP The key design decision in MiniCP is the one-to-one mapping between the theoretical foundations of constraint programming and the solver architecture. The existence of this mapping fundamentally simplifies the understanding of the implementation. The second design decision in MiniCP is the focus on extensibility and compositionality: It is reasonably simple to add new features to MiniCP and the interactions between these features are limited. The organization of this paper reflects these design decisions. The paper starts with a review of the foundations of constraint programming before focusing on its two main components: filtering and search. This is followed by a presentation of some more advanced features in filtering and search.

It is important to emphasize that MiniCP does *not* attempt to cover every aspect of constraint programming: This would defeat the purpose of this endeavor. Topics such as model reformulation and transformation, parallelization, richer variable types, e.g., set or continuous variables, learning-based constraint-programming solvers, copy-based solvers, soft constraints, and MDD-based approaches to name only a few are not discussed in this paper. This does not mean that they cannot be supported in MiniCP. Rather these topics could be the subject of future extensions or exploratory projects.

MiniCP is implemented in Java 8 to make it accessible to a large audience. The code makes extensive use of closures introduced in Java 8 through the concept of functional interfaces. Appendix A contains a review of Java closures for completeness. Although the focus in MiniCP has been on simplicity and clarity (see Table 1 for its size), its performance is reasonable. Appendix B provides some benchmark results to substantiate this statement.

Influences The design of MiniCP is primarily influenced by the constraint-programming systems cc(fd) [30], Comet [5,7], Objective-CP [27], and OscaR [18]. Some inspiration also came from the microkernel architecture introduced in [15]. Other solvers sharing similar designs are CP Optimizer [13], OR-Tools [17], [12], Mistral [?] and CHOCO [19]. Most of the implementation techniques in MiniCP were introduced in the constraint-programming system CHIP [24,1]. For brevity, the presentation only provides references when this is not the case. Readers can also consult the chapter on implementation in the handbook of constraint programming [21].

Outline The rest of this paper is organized as follows. Section 2 reviews the foundations of constraint programming and Section 3 provides a preview of MiniCP through three running examples. Section 4 covers the filtering component of constraints and Section 5 its search component. Sections 6 and 7 considers some advanced filtering and search functionalities. Section 8 discusses the teaching material and Section 9 concludes the paper.

2 Foundations of Constraint Programming

This section presents the foundations of MiniCP. It covers, at a high-level level of abstraction, key concepts of constraint programming that will be refined in the actual implementation of MiniCP. Section 2.1 defines constraint satisfaction problems, the class of problems addressed by constraint-programming solvers. Section 2.2 then introduces filtering algorithms, the fundamental computational building block of constraint programming. Section 2.3 specifies and shows how to implement constraint propagation in terms of filtering algorithms. Section 2.4 proposes the concept of branching scheme that forms the basis of the search component of MiniCP. Finally, Section 2.5 specifies the search algorithm implemented in the MiniCP solver.

Draft September 15, 2018

2.1 Constraint Satisfaction Problems

Constraint-programming systems are tools to solve constraint satisfaction problems. A constraint satisfaction problem (CSP) is defined in terms of variables, domains, and constraints. This section formalizes these concepts which form the core of the MiniCP solver. Many generalizations are possible and some are discussed later in the paper.

Definition 1 (Domain) A domain is a finite set of discrete values in $\mathbb{Z}$.

Domains are denoted by the letter $\mathcal{D}$ and their values by the letter v. Domains support a variety of operations including membership ($v \in \mathcal{D}$), lower bound ($\min(\mathcal{D}) = \min_{v \in \mathcal{D}} v$), and upper bound ($\max(\mathcal{D}) = \max_{v \in \mathcal{D}} v$). Domains inherit the total ordering of $\mathbb{Z}$.

Definition 2 (Decision Variable) A decision variable x is associated with a domain $\mathcal{D}$. It is instantiated (bound) if $|\mathcal{D}| = 1$.

Definition 3 (Constraint) A constraint c is a relation defined over a set of decision variables. $Vars(c)$ denotes the variables of constraint c.

Definition 4 (CSP) A CSP is a triplet $\langle X, \mathcal{D}, C \rangle$ where X is a set of decision variables, $\mathcal{D}$ is the Cartesian product of their domains, and C is a set of constraints defined over subsets of X. The domain of a decision variable x is denoted by $\mathcal{D}(x)$. Note that $\mathcal{D} = \mathcal{D}(x_0) \times \cdots \times \mathcal{D}(x_{n-1})$ when $X = \{x_0, \ldots, x_{n-1}\}$.

Solving a CSP amounts to assigning the decision variables to values in their domains so that all constraints are satisfied. For simplicity, the presentation often assumes an underlying CSP $\langle X, \mathcal{D}, C \rangle$ and refers to $\mathcal{D}$ as the domain of the CSP. Given two domains $\mathcal{D}_1$ and $\mathcal{D}_2$ over the same variables $\{x_0, \ldots, x_{n-1}\}$, the intersection $\mathcal{D}_1 \cap \mathcal{D}_2$ is defined as $\mathcal{D}_1(x_0) \cap \mathcal{D}_2(x_0) \times \ldots \times \mathcal{D}_1(x_{n-1}) \cap \mathcal{D}_2(x_{n-1})$.

Definition 5 (Candidate Solution) A candidate solution σ assigns to each decision variable x a value in its domain, i.e., $\sigma(x) \in \mathcal{D}(x)$.

Definition 6 (Constraint Valuation) The valuation of a constraint c with respect to a candidate solution σ is the Boolean value $c(\sigma)$ which stands for $c(\sigma(x_0), \ldots, \sigma(x_{n-1}))$ where $Vars(c) = \{x_0, \ldots, x_{n-1}\}$.

Definition 7 (Solution) A solution σ to a CSP $\langle X, \mathcal{D}, C \rangle$ is a candidate solution that satisfies all constraints, i.e., $\forall c \in C : c(\sigma)$. The set of solutions to a CSP $\langle X, \mathcal{D}, C \rangle$ is denoted by $\mathcal{S}(\langle X, \mathcal{D}, C \rangle)$.

Note that a solution to a CSP can also be viewed as a domain $\mathcal{D}$ that binds all variables, i.e., $|\mathcal{D}(x)| = 1$ for every variable x. These concepts can also be generalized to the case where an objective function must be minimized (or maximized).

Definition 8 (COP) A constraint optimization problem (COP) is a quadruplet $\langle X, \mathcal{D}, C, f \rangle$ in which f is a real function over a subset of variables $Vars(f) \subset X$.

Definition 9 (Function Valuation) The valuation of a function f with respect to a candidate solution σ is the value $f(\sigma)$ which stands for $f(\sigma(x_0), \ldots, \sigma(x_{n-1}))$ where $Vars(f) = \{x_0, \ldots x_{n-1}\}$.

Definition 10 (Optimal Solution) An optimal solution to a COP $\langle X, \mathcal{D}, C, f \rangle$ is a solution σ^* such that $\forall \sigma \in \mathcal{S}(\langle X, \mathcal{D}, C \rangle) : f(\sigma^*) \leq f(\sigma)$.

2.2 Filtering Algorithms

Filtering algorithms are the cornerstone of constraint programming. Their goal is to prune values from the variable domains without removing any solutions. Each constraint is associated with a filtering algorithm which receives domains for the constraint variables as input and returns new domains.

Definition 11 (Filtering Algorithm) A filtering algorithm $\mathcal{F}_c$ for a constraint c is a function from domain to domain satisfying

$$\forall \mathcal{D} : \mathcal{F}_c(\mathcal{D}) \subseteq \mathcal{D} \ \wedge \ \mathcal{S}(\langle Vars(c), \mathcal{D}, \{c\} \rangle) = \mathcal{S}(\langle Vars(c), \mathcal{F}_c(\mathcal{D}), \{c\} \rangle).$$

A filtering operator $\mathcal{F}_c$ is monotone if $\mathcal{D}_1 \subseteq \mathcal{D}_2 \Rightarrow \mathcal{F}_c(\mathcal{D}_1) \subseteq \mathcal{F}_c(\mathcal{D}_2)$.

Draft September 15, 2018

Algorithm 1: The Constraint-Propagation Algorithm.

Data: The CSP $\langle X, \mathcal{D}^0, C\rangle$
Result: The greatest fixpoint domain as defined in equation (1)

```
1 pruningNeeded ← true;
2 D ← D^0;
3 while pruningNeeded do
4     D^p ← F_C(D);
5     pruningNeeded ← D^p ≠ D;
6     D ← D^p;
```

Draft September 15, 2018

Constraint-programming solvers typically aim at enforcing some strong consistency properties on the output of a filtering algorithm. Bound and domain consistency are two such desirable properties. Bound consistency requires that, after filtering, the bounds of every variable belong to a solution of the constraint. Domain consistency requires that, after filtering, each value in the variable domains belong to a solution of the constraint. Domain consistency is the strongest property that can be enforced by a filtering algorithm.

Definition 12 (Bound Consistency) A constraint c over variables $Vars(c) = \{x_0, \ldots, x_{n-1}\}$ is bound-consistent wrt $\mathcal{D}$ if and only if, for every $i \in 0..n-1$, there exist values $v_j \in \mathcal{D}(x_j)$ $(0 \leq j < n : j \neq i)$ such that $c(v_0, \cdots, v_{i-1}, \min(\mathcal{D}(x_i)), v_{i+1}, \cdots, v_{n-1}) \wedge c(v_0, \cdots, v_{i-1}, \max(\mathcal{D}(x_i)), v_{i+1}, \cdots, v_{n-1})$ holds.

Definition 13 (Domain Consistency) A constraint c over variables $Vars(c) = \{x_0, \ldots, x_{n-1}\}$ is domain-consistent wrt $\mathcal{D}$ if and only if, for every $i \in 0..n-1$ and every value $v_i \in \mathcal{D}(x_i)$, there exist values $v_j \in \mathcal{D}(x_j)$ $(0 \leq j < n : j \neq i)$ such that $c(v_0, \cdots, v_{i-1}, v_i, v_{i+1}, \cdots, v_{n-1})$ holds.

Definition 14 (Consistent Filtering Algorithm) A filtering algorithm $\mathcal{F}_c$ for a constraint c is bound-consistent (resp. domain-consistent) if, for all domain $\mathcal{D}$, c is bound-consistent (resp. domain-consistent) wrt $\mathcal{F}_c(\mathcal{D})$.

Example 1 (Bound-Consistent Filtering Algorithm of $x = y + 1$.) Consider a constraint c of the form $x = y + 1$. A bound-consistent filtering algorithm returns the domains

$$\mathcal{F}_c(\mathcal{D})(x) = \{v \in \mathcal{D}(x) : \min(\mathcal{D}(y)) + 1 \leq v \leq \max(\mathcal{D}(y)) + 1\},$$
$$\mathcal{F}_c(\mathcal{D})(y) = \{v \in \mathcal{D}(y) : \min(\mathcal{D}(x)) - 1 \leq v \leq \max(\mathcal{D}(x)) - 1\}.$$

Example 2 (Domain-Consistent Filtering Algorithm of $x = y + 1$.) Consider a constraint c of the form $x = y + 1$. A domain-consistent filtering algorithm returns the domains

$$\mathcal{F}_c(\mathcal{D})(x) = \{v \in \mathcal{D}(x) : v - 1 \in \mathcal{D}(y)\},$$
$$\mathcal{F}_c(\mathcal{D})(y) = \{v \in \mathcal{D}(y) : v + 1 \in \mathcal{D}(x)\}.$$

It is useful to point that, while the domain-consistent filtering algorithm must be applied each time a value is pruned from the domain of x or y, this is not the case for the bound-consistent version. Indeed, for enforcing bound consistency, it suffices to apply the filtering when the bounds of x or y are updated. MiniCP implements this optimization by associating dedicated lists of constraints with variables as discussed in Section 4.3.

2.3 Constraint Propagation

The core of a constraint-programming solver is a propagation algorithm that applies filtering algorithms until no more value can be pruned from the variable domains. Algorithm 1 depicts the most basic constraint-propagation algorithm: It simply iterates the application of the filtering algorithm associated with each constraint until none of them reduces the domains. More formally, Algorithm 1 computes the greatest fixpoint of the operator $\mathcal{F}_C = \bigcap_{c \in C} \mathcal{F}_c(\mathcal{D})$. In other words, given a CSP $\langle X, \mathcal{D}, C\rangle$, the constraint propagation $\mathcal{F}$ returns a domain $\mathcal{D}^*$ defined by

$$\mathcal{D}^* = \max\{\mathcal{D}^p \subseteq \mathcal{D} : \mathcal{D}^p = \mathcal{F}_C(\mathcal{D})\}. \quad (1)$$

When the filtering algorithms are monotone, this greatest fixpoint is unique and hence the order in which the filters are applied has no impact on the results. The constraint propagation can also be viewed as

Algorithm 2: The Data-Driven Constraint-Propagation Algorithm.

Data: The CSP $\langle X, \mathcal{D}^0, C\rangle$
Result: The greatest fixpoint domain as defined in equation (1)
1 $Q \leftarrow C$;
2 $\mathcal{D} \leftarrow \mathcal{D}^0$;
3 **while** $|Q| > 0$ **do**
4 $\quad c = \text{dequeue}(Q)$;
5 $\quad \mathcal{D}^p \leftarrow \mathcal{F}_c(\mathcal{D})$;
6 $\quad V \leftarrow \{x \in Vars(c) : D^p(x) \neq \mathcal{D}(x)\}$;
7 $\quad$ **if** $|V| > 0$ **then**
8 $\qquad Q \leftarrow Q \cup \{c : |Vars(c) \cap V| > 0\}$;
9 $\quad \mathcal{D} \leftarrow \mathcal{D}^p$;

transforming a CSP $\langle X, \mathcal{D}, C\rangle$ into a CSP $\langle X, \mathcal{D}^*, C\rangle$ such that $\mathcal{S}(\langle X, \mathcal{D}, C\rangle) = \mathcal{S}(\langle X, \mathcal{D}^*, C\rangle)$ and $\mathcal{D}^*$ satisfies Equation 1. If $|\mathcal{D}^*| = 0$, the CSP has no solution. If $|\mathcal{D}^*| = 1$, then $\mathcal{D}^*$ is a solution. Otherwise, no conclusion can be drawn. When $|\mathcal{D}^*| = 0$, we say that the constraint propagation fails or is a failure. When $|\mathcal{D}^*| = 1$, we say that the propagation succeeds or is a success.

Algorithm 1 is rather naive as it applies all filtering algorithms each time the domain of a variable is pruned. In practice, constraint-programming solvers only reconsider the filtering algorithms for constraints whose variables have seen their domains reduced. This is captured in Algorithm 2 which is organized around a queue of constraints. Initially, the queue Q contains all constraints and each iteration pops a constraint from Q and applies its filtering algorithm. It then enqueues all constraints that have at least one variable whose domain has been pruned. The algorithm terminates when the queue is empty.[1] The MiniCP implementation goes one step further by also avoiding to enqueue constraints whose filtering algorithms will not prune the domains as discussed earlier. This is covered in detail in Section 4.3 where it is shown that each variable is associated with several lists of constraints.

Draft September 15, 2018

2.4 Branching

Constraint propagation may fail, succeed, or be inconclusive. In the last case, to make further progress, constraint-programming solvers typically partition the CSP into a set of simpler CSPs. Some solvers offer substantial flexibility to express how to branch, while others do not give any control to users. MiniCP uses the concept of *branching scheme* to express the decomposition process.

Definition 15 (Branching Scheme) A branching scheme for a CSP $\langle X, \mathcal{D}, C\rangle$ is a set of CSPs $\{\langle X, \mathcal{D}, C \cup \{c_0\}\rangle, \ldots, \langle X, \mathcal{D}, C \cup \{c_{k-1}\}\rangle\}$ such that

$$\mathcal{S}(\langle X, \mathcal{D}, C\rangle) = \bigcup_{i \in 0..k-1} \mathcal{S}(\langle X, \mathcal{D}, C \cup \{c_i\}\rangle)$$

and, for all i, j with $0 \le i < j < k$,

$$|\mathcal{S}(\langle X, \mathcal{D}, C \cup \{c_i\}\rangle) \cap \mathcal{S}(\langle X, \mathcal{D}, C \cup \{c_j\}\rangle)| = 0.$$

Note that, to specify a branching scheme, it suffices to specify the constraints $\{c_0, \ldots, c_{k-1}\}$.

Example 3 (Minimum Value Branching) The minimun value branching selects a free variable $x \in X$ and uses two constraints: $x = \min(\mathcal{D}(x))$ and $x \neq \min(\mathcal{D}(x))$.

Example 4 (Dichotomic Value Branching) The dichotomic value branching selects a free variable $x \in X$ and uses two constraints: $x \le \text{mid}(\mathcal{D}(x))$ and $x > \text{mid}(\mathcal{D}(x))$, where $\text{mid}(\mathcal{D})$ is the mid-point value in $\mathcal{D}$.

2.5 Search

It is now possible to specify the search algorithm underlying MiniCP as a simple recursive algorithm implementing a depth-first search with chronological backtracking. Procedure `CPSearch` is depicted

[1] When all the filtering algorithms are domain-consistent, Algorithm 2 is equivalent to the AC-3 algorithm in [14].

Algorithm 3: The Abstract Version of the MiniCP Search Algorithm.

```
   Data: The CSP ⟨X, D, C⟩
   Result: CPSearch(⟨X, D, C⟩) = S(⟨X, D, C⟩)
 1 D* ← F(⟨X, D, C⟩) ;
 2 if |D*| = 0 then
 3   return ∅ ;
 4 else if |D*| = 1 then
 5   return {D*} ;
 6 else
 7   (c_0, ··· , c_{k−1}) ← branch(⟨X, D*, C⟩);
 8   return ⋃_{i=1}^{k} CPSearch(⟨X, D*, C ∪ {c_i}⟩);
```

in Algorithm 3; It takes a CSP as input and returns its solutions. The algorithm first performs constraint propagation (line 1). If the propagation fails, the algorithm returns no solution (lines 2–3). If the propagation succeeds, it returns the solution (lines 4–5). Otherwise, the algorithm applies the branching scheme to obtain a set of constraints (line 7). The resulting CSPs are solved recursively and the algorithm returns the union of their solutions. The main difference between this abstract version and the actual implementation in MiniCP is the way the domains and the constraints are updated which is discussed in Section 4.4.

Note that MiniCP solves optimization problems as a sequence of feasibility problems. Each time a solution with objective value f^* is found, MiniCP searches for a feasible solution whose objective improves upon f^*. The actual implementation, which also avoids redundant computations, is discussed in Section 5.4.

3 A Preview of MiniCP

This section presents some simple MiniCP programs. Their main purpose is to provide a clear link between the theoretical foundations and the implementation. These programs will introduce the concrete conterparts to the abstract concepts presented in Section 2: The decision variables, the domains, the constraints, the branching scheme, and the search. Although these constraint programs are relatively simple, they convey the essence of MiniCP and its implementation. More sophisticated techniques are presented later in the paper.

3.1 The N-Queens Problem

The first MiniCP program solves the well-known queens problem which amounts to placing n queens on an $n \times n$ checkerboard so that no two queens can attack each other, i.e., no two queens are on the same row, column, or diagonals.

The MiniCP model uses a simple encoding that associates a decision variable q_i with each column $i \in \{0, \ldots, n-1\}$ to represent the row of the queen placed in this column. By virtue of the encoding, it is only necessary to impose constraints that ensure that no queens are on the same row and diagonals, which can be expressed as

$$\begin{array}{l} \forall\, i, j \in 0..n-1 \wedge i < j : q_i \neq q_j \\ \forall\, i, j \in 0..n-1 \wedge i < j : q_i - i \neq q_j - j \\ \forall\, i, j \in 0..n-1 \wedge i < j : q_i + i \neq q_j + j \end{array}$$

The MiniCP program is presented in Listing 1 and contains three main parts: the declaration of the decision variables (line 3), the constraint specification (lines 5–10), and the branching scheme (lines 12–28). Line 2 creates a CP solver `cp` and line 3 creates the array q of n decision variables, each with a domain $\{0..n-1\}$. Lines 5–10 create the binary disequations and post them to the CP solver.

The branching scheme in lines 12–28) is given as a closure (see Appendix A for a review of Java closures) that will be applied repeatedly during the search as specified in Algorithm 3. The branching has two main parts: the selection of the variable to assign (lines 13–18) and the creation of the branching constraints (lines 19–27). The variable selection is extremely simple: The first free variable is selected. The code simply iterates over all variables in the array and computes the index `idx` of this free variable. More complicated variable selections (e.g., selecting the free variable with the smallest domain) can be easily implemented in a similar way. The remaining of the branching scheme generates branching

Draft September 15, 2018

Listing 1: A MiniCP Program for the N-Queens Problem

```
1  int n = 8; // number of queens and size of board
2  Solver cp  = Factory.makeSolver();
3  IntVar[] q = Factory.makeIntVarArray(cp,n,0,n-1);
4
5  for (int i = 0; i < n; i++)
6     for (int j = i+1; j < n; j++) {
7        cp.post(Factory.notEqual(q[i], q[j]));
8        cp.post(Factory.notEqual(q[i], q[j], i-j));
9        cp.post(Factory.notEqual(q[i], q[j], j-i));
10    }
11
12 DFSearch dfs = Factory.makeDfs(cp, () -> {
13    int idx = -1; // index of the first variable that is not bound
14    for (int k = 0; k < q.length; k++)
15       if (q[k].size() > 1)  {
16          idx = k;
17          break;
18       }
19    if (idx == -1)
20       return new Procedure[0];
21    else {
22          IntVar qi = q[idx];
23          int v = qi.min();
24          Procedure left  = () -> Factory.equal(qi, v);
25          Procedure right = () -> Factory.notEqual(qi, v);
26          return new Procedure[]{left,right};
27    }
28 });
29 dfs.solve();
```

constraints that are represented as an array of procedures (i.e., `void` to `void` closures). If all variables are bound, then the branching scheme returns an empty array (line 20) to notify the search it is a solution. Otherwise, the code implements the minimum value branching specified in Example 3. In Lines 24–25, the branching scheme defines two closures which will create the branching constraints when called and these closures are inserted in the array of branching decisions in line 26. The branching scheme is passed as a parameter to a depth-first search object in Line 12. The depth-first search is performed during the `solve` call in line 29 and computes all solutions. To print the solution, it suffices to add the code

```
1 dfs.onSolution(() ->
2        System.out.println("solution:" + Arrays.toString(q))
3 );
```

before line 29.

The binary disequations in lines 5–10 can be replaced by three global constraints:

```
1 cp.post(Factory.allDifferentAC(q));
2 cp.post(Factory.allDifferentAC(Factory.makeIntVarArray(cp,n,i -> Factory.minus(q[i],i))));
3 cp.post(Factory.allDifferentAC(Factory.makeIntVarArray(cp,n,i -> Factory.plus(q[i],i))));
```

Global constraints are a fundamental concept in constraint programming as they enable to capture sub-structures arising in many applications. These global constraints can then be associated with dedicated filtering algorithms that exploits the properties of these substructures. In the queens problem, the global constraints express that a collection of variables must take different values. For instance, Line 1 specifies that all variables in array `q` must take different values. Line 2 above deserves some explanations. The sub-expression

```
1 Factory.makeIntVarArray(cp,n,i -> Factory.minus(q[i],i)))
```

makes use of the function of signature

```
IntVar[] makeIntVarArray(int n,Function<Integer,IntVar> body)
```

which creates an array of size `n` whose variables are obtained using the closure parameter. In the above instruction, the function `Factory.minus(q[i],i)` returns an integer variable, say `x[i]`, whose values are subject to the constraint `x[i] = q[i] - i`. As a result, the second `allDifferent` constraint specifies that the expressions $q_i - i$ $(0 \leq i < n)$ evaluate to different values.

Draft September 15, 2018

Listing 2: A MiniCP Program for the Magic Series Problem

```
1 import static minicp.cp.Factory.*;
2
3 int n = 8; // size of the series
4 Solver cp = makeSolver();
5 IntVar[] s = makeIntVarArray(cp,n,0,n);
6
7 for(int i = 0; i < n; i++)
8    cp.post(sum(makeIntVarArray(0, n - 1, j -> isEqual(s[j], i)), s[i]));
9 cp.post(sum(s,n));
10 cp.post(sum(makeIntVarArray(0, n - 1, i -> mul(s[i], i)), n));
11
12 DFSearch dfs = makeDfs(cp, () -> {
13     // similar to n-queens search but on variables s[i]
14 });
15
16 dfs.solve();
```

Draft September 15, 2018

The MiniCP code makes heavy use of the factory design pattern [?] to create solvers, variables, and constraints using static methods (e.g., `makeSolver`, `makeIntVarArray`, `sum`, `isEqual`, `mul`, `makeDfs`) of the `Factory` class. It is possible to use the Java `import static` instruction to make the factory calls implicit as illustrated in the next MiniCP example.

3.2 The Magic Series Problem

A series $S = (s_0, s_1, \ldots, s_{n-1})$ is magic if s_i represents the number of occurrences of i in S. Listing 2 gives a MiniCP program for finding magic series. The core of the MiniCP model is the constraints

$$\sum_{j=0}^{n-1} \mathbb{1}(s_j = i) = s_i \quad (0 \leq i < n)$$

where $\mathbb{1}$ denotes the indicator function. The last two constraints,

$$\sum_{j=0}^{n-1} s_j = n$$

and

$$\sum_{j=0}^{n-1} j \times s_j = n$$

express properties of magic series and help reduce the search space.

The MiniCP program is a direct encoding of these constraints. It makes use of the `sum(a,s)` constraint that holds if the elements in array `a` sum to `s`. It also uses the `isEqual(x,v)` indicator function that returns 0-1 variable whose value is 1 if `x` is equal to `v` and 0 otherwise. The use of indicator functions is often called reification in constraint programming and is explained in Section 6.

The impact of the last two constraints is significant: Without them, the search for a magic series of length 200 requires 32,430 choices. With them, only 400 choices are needed.

3.3 A Quadratic Assignment Problem

This section considers an assignment problem where a set of n facilities must be assigned to n different locations. The distance between two locations l and m is given by $d_{l,m}$ and each pair of facilities (i,j) is associated with a weight $w_{i,j}$. The goal of the assignment is to minimize the sum of the weighted distances between the locations of each pair of facilities.

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} d_{x_i,x_j} \cdot w_{i,j}$$

Listing 3: MiniCP model for the QAP

```
1  Solver cp  = makeSolver();
2  IntVar[] x = makeIntVarArray(cp,n,0,n-1);
3
4  cp.post(allDifferent(x));
5  IntVar[] weightedDist = new IntVar[n*n]; // build the objective function
6  int k = 0;
7  for (int i = 0; i < n; i++)
8     for (int j  = 0; j < n; j++) {
9         IntVar dij = element(d,x[i],x[j]);
10        weightedDist[k++] = mul(w[i][j],d_ij);
11    }
12 Objective obj = cp.minimize(sum(weightedDist));
13 DFSearch dfs = makeDfs(cp,  () -> {
14     // similar to n-queens search but on variables x[i]
15 });
16 dfs.optimize(obj);
```

Listing 4: Integer Variable Interface

```
1  public interface IntVar {
2    Solver getSolver();
3    int min();
4    int max();
5    int size();
6    boolean contains(int v);
7    boolean isBound();
8
9    void remove(int v);
10   void assign(int v);
11   void removeBelow(int v);
12   void removeAbove(int v);
13
14   void propagateOnDomainChange(Constraint c);
15   void propagateOnBoundChange(Constraint c);
16   void propagateOnBind(Constraint c);
17 }
```

The model is given in Listing 3. It assumes the presence of two inputs: matrix `w` that specifies the weights and matrix `d` that specifies the distances. It makes use of the `element` constraint [?] that allows decision variables to index arrays and matrices and whose implementation is discussed in Section 6.4. More specifically, `element(d,x[i],x[j])` returns a variable whose value is equal to the expression $d[x[i]][x[j]]$.

To build the objective function, the MiniCP program creates an auxiliary variable dij for the distance between two facilities i and j in the loop spanning lines 7–11. The same loop also accumulates in array `weightedDist` all the weighted distance. Line 12 then specifies the objective as the minimization of the sum of the weighted distances.

4 The Constraint Propagation Implementation

This section describes the MiniCP implementation of constraint propagation. Section 4.2 proposes one possible implementation for domains and Section 4.3 describes the implementation of variables. Section 4.4 discusses how to implement constraints and Section 4.5 depicts the constraint-propagation algorithms. Since many of these concepts refer to each other, the section starts with a number of interfaces.

4.1 Interfaces

The Variable Interface The interface for integer variables is given in Listing 4 and offers three classes of methods. First, it provides a number of accessors to query the variable domain. Second, it offers a number

Draft September 15, 2018

Listing 5: Constraint Interface

```
1 public interface Constraint {
2   void post();
3   void propagate();
4   void setScheduled(boolean scheduled);
5   boolean isScheduled();
6   void setActive(boolean active);
7   boolean isActive();
8 }
```

Listing 6: Objective Interface

```
1 public interface Objective {
2     void tighten();
3 }
```

Listing 7: The Domain Interface

```
1 public interface IntDomain {
2   public int min();
3   public int max();
4   public int size();
5   public boolean isBound();
6   public boolean contains(int v);
7   public void remove(int v,DomainListener l);
8   public void removeAllBut(int v,DomainListener l);
9   public void removeBelow(int v,DomainListener l);
10  public void removeAbove(int v,DomainListener l);
11 }
```

Draft September 15, 2018

of mutators to reduce the variable domain (i.e., removing a value, assigning a value, removing values below a certain threshold, and removing values above a certain threshold). Finally, it provides a number of methods to link variables and constraints. More specifically, method `propagateOnDomainChange(c)` specifies that constraint `c` must be propagated each time the domain of the variable is updated, method `propagateOnBoundChange(c)` specifies that constraint `c` must be propagated each time one of its bounds is updated, and method `propagateOnBind(c)` indicates that constraint `c` must be propagated when the variable is bound.

The Constraint Interface The constraint interface is given in Listing 5. The main methods are `post` and `propagate`. Method `post` is called *once* when the constraint is added to the MiniCP solver. It typically initializes internal data structures, links variables and constraints, and performs the first propagation step. Method `propagate` is the core of the implementation: It applies the filtering algorithm of the constraint and is called *repeatedly*. The remaining methods are used for optimizing constraint propagation and are discussed in Section 4.5

The Objective Interface The objective interface is given in Listing 6. The sole method `tighten` is meant to be called when the solver produces a solution to a constraint optimization problem compelling the solver to produce a next solution whose quality strictly exceeds that of the incumbent. If the COP is a minimization problem, the `tigthen` method would pickup the primal bound as value of the objective function and then impose the requirement that the objective should be strictly less than this new primal bound.

The Domain Interface The interface of domains is given in Listing 7. The first set of methods return the minimum and maximum values in the domain, the domain size, whether the variable is bound, and whether a value v belongs to the domain. The remaining four methods respectively remove, from the domain, value v, all values but v, all values smaller than v, and all value greater than v. These last four methods receive as argument a domain listener, whose interface is given in Listing 8, as proposed in [29].

Listing 8: The Domain Listener Interface

```
1 public interface DomainListener {
2   public void empty();
3   public void bind();
4   public void change();
5   public void changeMin();
6   public void changeMax();
7 }
```

Listing 9: The Solver Interface

```
1 public interface Solver {
2   void post(Constraint c);
3   Objective minimize(IntVar x);
4   Objective maximize(IntVar x);
5   void schedule(Constraint c);
6   void fixPoint();
7 }
```

Draft September 15, 2018

A domain listener allows an (arbitrary) object to be notified of various events on the domain, namely, when the domain becomes empty (`empty`), when it has only one element (`bind`), when it has changed (`change`), or when its smallest (`changeMin`) or largest (`changeMax`) value has changed. As mentioned in Section 2.2, this information is useful to decide whether a constraint needs to be propagated after a domain update.

Solver Interface The `Solver` interface is shown in Listing 9. Method `post` is used to register constraints. The `minimize` and `maximize` methods are available to state an objective function. Method `schedule` is called to schedule a constraint for propagation and method `fixPoint` implements algorithm 2.

4.2 A Domain Implementation

This section proposes a domain implementation in terms of sparse sets [?]. The implementation of the domain is rather generic however and other set representations can easily be used instead.

Listing 10 presents the core of the domain implementation in terms of a sparse set. The constructor creates the sparse set and the domain accessors accessors simply delegate to the sparse set. The `remove` method is more interesting: Its role is not only to delegate the removal to the sparse set but also to notify the domain listener about which domain events are arising due to the removal. Lines 13–14 test whether the minimum or maximum value is removed, and line 15 delegates the removal to the sparse set. The rest of the method code simply performs the proper calls on the domain listener: It calls `empty` if the domain becomes empty, `changeMin` and `changeMax` if the minimum or maximum values are removed, `change` if a value is removed, and `bind` if there is a single value left in the domain. The remaining methods for value removals can be implemented similarly.

It remains to present the sparse set implementation which represents subset of numbers between a lower bound L and an upper bound U. For simplicity, the presentation assumes that $L = 0$: It is easy to add a shift factor to deal with the more general case. The sparse set uses two arrays, `values` and `indices`, and an integer `size` to track how many elements are in the set. Array `values` contains a permutation of the values $0..U$, while array `indices` keeps track of the positions of the elements in array `values`. In other words, it holds that

$$\texttt{values}[\texttt{indices}[i]] = i.$$

Figure 1 depicts the representation of set $\{0, \ldots, 8\}$ after initialization and Figure 2 shows the representation when values 4 and 6 have been removed. Value 4 is removed by swapping its value in array `values` with the last element of the array, updating array `indices` appropriately, and decrement `size` by one. Removing value 6 amounts to swapping 6 with the last present element of array `values`, i.e., 7, and updating the indices and `size`.

Listing 10: A Domain Implementation

```
1 public class SparseSetDomain implements IntDomain {
2     private SparseSet spset;
3     public SparseSetDomain(StateManager sm,int min,int max) {
4         spset = new SparseSet(sm,max-min+1,min);
5     }
6     public int min()  { return spset.min(); }
7     public int max()  { return spset.max(); }
8     public int size() { return spset.size(); }
9     public boolean contains(int v) { return spset.contains(v); }
10    public boolean isBound() { return spset.size() == 1; }
11    public void remove(int v, DomainListener l) {
12        if (spset.contains(v)) {
13            boolean maxChanged = max() == v;
14            boolean minChanged = min() == v;
15            spset.remove(v);
16            if (spset.size() == 0) l.empty();
17            l.change();
18            if (maxChanged) l.changeMax();
19            if (minChanged) l.changeMin();
20            if (spset.size() == 1) l.bind();
21        }
22    }
23    public void removeBelow(int value,DomainListener l) {...}
24    public void removeAbove(int value,DomainListener l) {...}
25    public void removeAllBut(int v,DomainListener l)
26 }
```

Draft September 15, 2018

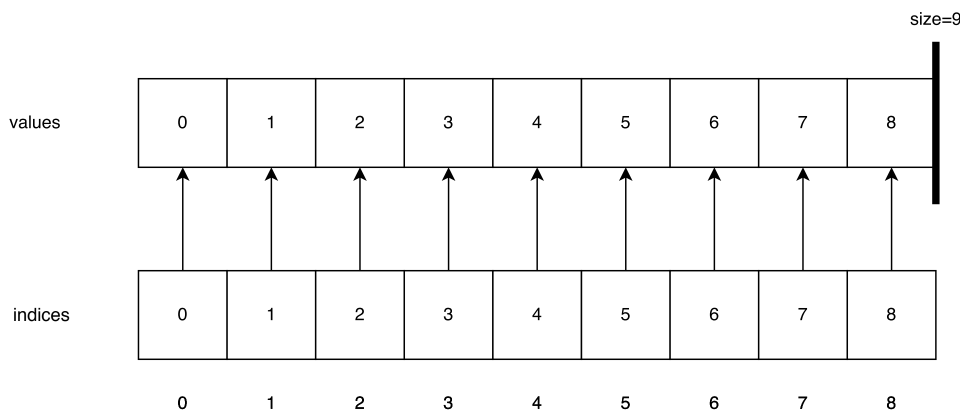

Fig. 1: A Sparset Set of 9 Values at Initialization.

All the elements present in the set are between indices 0 and `size-1`. Finding out if an element is the set takes constant time: It suffices to consult array `indices` and test whether its position is smaller than `size`. Removing a value also takes constant time as highlighted by the discussion above. Iterating over the elements of the set takes linear time in the number of elements in the set. Removing all values but v amounts to swapping v with the element in the first position, updating the entries in `indexes` to reflect the new positions, and updating `size` to 1. Removing all the values below (or above) a given threshold v take time $\mathcal{O}(|\Delta|)$ where Δ is the set of removed values: It suffices to apply the remove operation for each value that needs to be removed.

4.3 The Variable Implementation

Listing 11 presents the implementation of integer variables. The instance variables include a reference to the solver, a reference to the domain and three stacks of constraints based on the abstract data type `StateStack` (See section 5.3 for details on this data structure). Each stack corresponds to an event type, specifying when the constraints must be scheduled for propagation. For instance, the *onBind* stack stores the constraints to be propagated when the variable is bound.

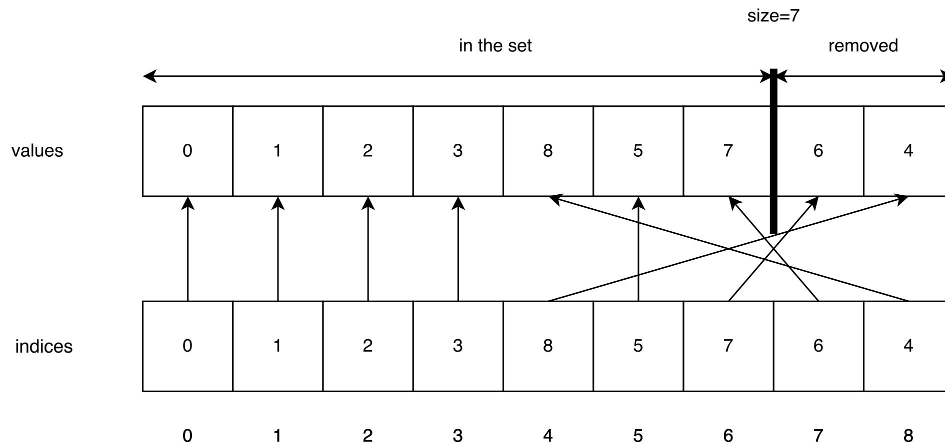

Fig. 2: Sparset Set

Listing 11: IntVar Implementation

```
public class IntVarImpl implements IntVar {
    private Solver cp;
    private IntDomain domain;
    private StateStack<Constraint> onDomain;
    private StateStack<Constraint> onBind;
    private StateStack<Constraint> onBounds;

    private DomainListener domListener = new DomainListener() {
        public void empty() { throw InconsistencyException.INCONSISTENCY;}
        public void bind()  { scheduleAll(onBind); }
        public void change()    { scheduleAll(onDomain); }
        public void changeMin() { scheduleAll(onBounds); }
        public void changeMax() { scheduleAll(onBounds); }
    };
    private void scheduleAll(Stack<Constraint> constraints) {
        for (int i = 0; i < constraints.size(); i++)
            cp.schedule(constraints.get(i));
    }

    public IntVarImpl(Solver cp, int min, int max) {
        if (min > max)
            throw new InvalidParameterException("empty domain");
        this.cp = cp;
        domain    = new Domain(cp,min,max);
        onDomain  = new Stack<>(cp);
        onBind    = new Stack<>(cp);
        onBounds  = new Stack<>(cp);
    }
    public int min()                { return domain.min();}
    public int max()                { return domain.max();}
    public int size()               { return domain.size();}
    public boolean contains(int v)  { return domain.contains(v);}
    public boolean isBound()        { return domain.size() == 1;}

    public void remove(int v)       { domain.remove(v, domListener);}
    public void assign(int v)       { domain.removeAllBut(v,domListener);}
    public void removeBelow(int v)  { domain.removeBelow(v,domListener); }
    public void removeAbove(int v)  { domain.removeAbove(v,domListener); }

    public void propagateOnDomainChange(Constraint c)  { onDomain.push(c); }
    public void propagateOnBind(Constraint c)          { onBind.push(c); }
    public void propagateOnBounds(Constraint c)        { onBounds.push(c); }
}
```

Lines 29–33 provide accessors to the variable domain, while lines 35–38 present the methods to remove values from the domain. Lines 40–42 describe methods to link constraints to the variable. The constructor creates the domain and the constraint stacks.

Draft September 15, 2018

Listing 12: The Class `Abstract Constraint`

```java
public abstract class AbstractConstraint implements Constraint {
    final protected Solver cp;
    // Any additional state required to implement the two optimizations.
    public AbstractConstraint(Solver cp) {
        this.cp = cp;
        ...
    }
    public void post()                            {}
    public void propagate()                       {}
    public void setScheduled(boolean scheduled)  { ... }
    public boolean isScheduled()                  { ... }
    public void setActive(boolean active)        { ... }
    public boolean isActive()                     { ... }
}
```

The remove methods are the most interesting aspect of the variable implementation. They delegate the removals to the domain, passing a domain listener as parameter. The methods of the listener are called depending upon what events are arising in the domain, e.g., method `empty` is called when the domain is empty and method `change` when the domain is updated. The implementation of the domain listener is shown in lines 8–14. Method `empty` throws an exception, indicating a failure of the propagation, which will be caught in the search implementation. The remaining methods simply schedule the relevant constraints for propagation. For instance, method `bind` schedules all constraints in the stack `onBind`.

Example 5 Consider a variable x with domain $\{1, 2, 3, 4, 5, 6\}$ and the result of an invocation of the `remove(int v)` method on value 3. The computation can be summarized as follows.

1. The removal is delegated to its domain (line 35). The domain method receives the value to remove (3) and the domain listener.
2. The `change` method of the listener is called (line 11) and consequently all the constraints present in `onDomain` container are scheduled for propagation (method `scheduleAll` spanning lines 15–18).

If the initial domain is $\{3, 4\}$, the domain would become a singleton $\{4\}$ and therefore the `bind` method of the listener would also be called to propagate the constraint in the `onBind` container. Finally, if the initial domain $\mathcal{D}(x)$ is the singleton $\{3\}$, the removal of 3 would make the domain empty, in which case the `empty` method of the listener would be called, which will throw an exception.

Observe how the domain listener modularizes the logic to react to domain events. By changing or upgrading the domain listener, it is possible to implement advanced features such as views.

4.4 The Implementation of Constraints

Constraints implement the `Constraint` interface in Listing 5 and are subclasses of `AbstractConstraint` shown in Listing 12. Class `AbstractConstraint` factorizes the implementation of methods that accelerate the fixpoint algorithm: These are discussed in Section 4.5. Subclasses override the `post` and `propagate` methods which are called when constraints are first posted to the solver and when constraints must be propagated respectively. During calls to `post` and `propagate`, exceptions of type `InconsistencyException` are raised each time a failure is encountered (e.g., a variable domain becomes empty). Since filtering algorithms exploit the semantics of constraints, it is easier to present examples to illustrate the implementation of constraints.

Example 6 ($x \leq y$) Consider the implementation of $x \leq y$. The inference rules are

1. $\max(\mathcal{D}(x)) \leftarrow \min\{\max(\mathcal{D}(x)), \max(\mathcal{D}(y)))\}$
2. $\min(\mathcal{D}(y)) \leftarrow \max\{\min(\mathcal{D}(y)), \min(\mathcal{D}(y)))\}$

The constraint implementation is given in Listing 13. The `post` method registers the constraint and links it to x and y with *boundChange* events, since no filtering takes place when removing values in the middle of the domain. The `propagate` method implements the filtering. Line 16 implements a simple activation optimization. If the constraint trivially holds, i.e., if $\max(\mathcal{D}(x)) \leq \min(\mathcal{D}(y))$, the constraint is disabled since no filtering will take place in subsequent computations.

Draft September 15, 2018

Listing 13: LessOrEqual Constraint

```
public class LessOrEqual extends AbstractConstraint { // x <= y
    private final IntVar x,y;
    public LessOrEqual(IntVar x, IntVar y) {
        super(x.getSolver());
        this.x = x;
        this.y = y;
    }
    @Override public void post() {
        x.propagateOnBoundChange(this);
        y.propagateOnBoundChange(this);
        propagate();
    }
    @Override public void propagate() {
        x.removeAbove(y.max());
        y.removeBelow(x.min());
        setActive(x.max() > y.min());
    }
}
```

Listing 14: NotEqual Constraint

```
public class NotEqual extends AbstractConstraint {
    private final IntVar x, y;
    private final int c;
    public NotEqual(IntVar x, IntVar y, int c) { // x != y + c
        super(x.getSolver());
        this.x = x;this.y = y;this.c = c;
    }
    public NotEqual(IntVar x, IntVar y) { this(x,y,0);}
    @Override public void post()  {
        if (y.isBound())
            x.remove(y.getMin() + c);
        else if (x.isBound())
            y.remove(x.getMin() - c);
        else {
            x.propagateOnBind(this);
            y.propagateOnBind(this);
        }
    }
    @Override public void propagate()  {
        if (y.isBound())
            x.remove(y.getMin() + c);
        else
            y.remove(x.getMin() - c);
    }
  }
```

Example 7 ($x \neq y + c$) Consider the implementation shown in Listing 14 of $x \neq y + c$ where c is a constant in $\mathbb{Z}$. The constraint was used in the queens program (Listing 1). The `post` method performs a simple case analysis. If either x or y is already bound, it removes the corresponding value from the domain of the other variable. Otherwise, it links the constraint with the variables with `bind` events. The `propagate` is invoked when one of the variables becomes bound. It removes the corresponding value from the domain of the other variables or fails if that variable is bound to that value.

4.5 The Implementation of Constraint Propagation

Class `MiniCP` in Listing 15 is the core of the constraint propagation and its `fixpoint` method implements Algorithm 2. The implementation is organized around a queue of constraints to propagate. Method `post` posts a constraint and performs constraint propagation. Method `fixPoint` (lines 13–23) pops constraints from the propagation queue and propagates them until the queue is empty or a failure occurs. In case of a failure, the exception is caught, the queue is emptied, and the exception is thrown again to

Draft September 15, 2018

Listing 15: Solver Implementation

```
1  public class MiniCP implements Solver {
2     private Queue<Constraint> propagationQueue = new ArrayDeque<>();
3     public void post(Constraint c) {
4         c.post();
5         fixPoint();
6     }
7     public void schedule(Constraint c) {
8         if (c.isActive() && !c.isScheduled()) {
9             c.setScheduled(true);
10            propagationQueue.add(c);
11        }
12    }
13    public void fixPoint() {
14        try {
15          while (propagationQueue.size() > 0)
16            propagate(propagationQueue.remove());
17        }
18        catch (InconsistencyException e) { // clear the propagation queue
19          while (propagationQueue.size() > 0)
20              clear(propagationQueue.remove());
21          throw e;
22        }
23    }
24    private void propagate(Constraint c) {
25        c.setScheduled(false);
26        if (c.isActive())
27            c.propagate();
28    }
29    private void clear(Constraint c)     { c.setScheduled(false);}
30 }
```

communicate it to the search. The propagation makes sure that a constraint is not pushed in the queue if it is already in it, using the `schedule` methods of the constraints. More precisely, method `schedule` (lines 7–12) checks whether the constraint is already scheduled and method `propagate(Constraint c)` (lines 24 to 28) resets the scheduled flag before calling the propagation provided that the constraint has not been deactivated. The flag is also reset in case of failure.

Unlike Algorithm 2, the implementation does not copy the domains of the variables (e.g., like in line 5 of Algorithm 2) but modifies them *in place*. These domains will be restored during backtracking as discussed in Section 5.2.

5 The Search Implementation

This section describes how to implement Algorithm 3. Observe that Algorithm 3 does not prescribe any search strategy: Different ordering policies for its queue Q lead to distinct search strategies. The MiniCP implementation is based on depth-first search, which is typical for constraint programming and is memory-efficient.

The main topic of this section is state management, the rest of the implementation being direct. As mentioned in Section 4.5, constraint propagation in MiniCP updates variable domains in place. As a result, in a first approximation, the variable domains after each fixpoint represent the state of the computation. When branching, MiniCP should thus save these domains in order to restore them in case of a failure so that the next branch is performed on the proper state. The state management is completely separated from the search itself and encapsulated in a state manager class. It is also important to mention that the MiniCP implementation exploits the last-in/first-out nature of depth-first search to optimize state management. Other search explorations may not benefit from similar optimizations.

The rest of this section is organized as follows. Section 5.1 presents the depth-first search implementation. Section 5.2 describes the state management and gives two possible implementations based on copying and trailing. The trailing state management strategy can be viewed as a optimized lazy implementation of the copying strategy. Section 5.3 revisits the domain implementation.

Draft September 15, 2018

Listing 16: The State Manager Interface

```
1 public interface StateManager {
2    void withNewState(Procedure body);
3    void saveState();
4    void restoreState();
5    StateInt makeStateInt(int initValue);
6    StateBool makeStateBool(boolean initValue);
7 }
```

Listing 17: Core DFS Skeleton

```
1 public class DFS {
2    private StateManager sm;
3    private Supplier<Procedure[]> branchingScheme;
4    public DFS(StateManager sm,Supplier<Procedure[]> b) {
5       this.sm = sm;branchingScheme = b;
6    }
7    public void solve() {
8       sm.withNewState( () -> {
9         dfs();
10       });
11    }
12    public void dfs() {
13       Procedure[] branches = branchingScheme.call();
14       if (branches.length == 0)
15          notifySolution();
16       else
17          for (b : branches) {
18             sm.withNewState( () -> {
19                try {
20                   b.call();
21                   dfs();
22                }
23                catch(InconsistencyException e) {}
24             });
25          }
26    }
27 }
```

Listing 18: The Equal Method

```
1 public class Factory {
2    ...
3    static public void equal(IntVar x, int v) {
4         x.assign(v);
5         x.getSolver().fixPoint();
6    }
7    ....
8 }
```

5.1 The Depth-First Search Implementation

Listing 17 depicts the implementation of depth-first search using the state manager interface presented in Listing 16. The important method at this stage is method `withNewState` which, informally speaking, executes a closure in a new state which is a copy of the current state. The depth-first search receives as input a state manager and a branching scheme, i.e., a closure that returns an array of branches when called. Method `solve` executes the depth-first search with a new state. Method `dfs` is the core of the search and is similar to Algorithm 3. It first applies the branching scheme to obtain an array of branches. If the array is empty, it means, in all the examples previewed in this paper, that the variables are all bound and the search completes by notifying that a solution has been found. Otherwise, the search iterates over all branches. For each of them, it creates a new state, applies the branch, and performs a

Draft September 15, 2018

Listing 19: StateInt

```
1 public interface StateInt {
2     int setValue(int v);
3     int value();
4 }
```

Listing 20: StateManager and StateInt Manipulation

```
1  StateInt a = sm.makeStateInt(7);
2  StateInt b = sm.makeStateInt(13);
3
4  sm.saveState();            // record current state a=7, b=13
5     a.setValue(6)
6     a.setValue(11);
7     sm.saveState();         // record current state a=11 b=13
8        a.setValue(4);
9        b.setValue(9);
10    sm.restoreState();  // now a=11, b=13
11 sm.restoreState();     // now a=7, b=13
```

Draft September 15, 2018

recursive call. The searches described in this paper use methods `equal` and `notEqual` and it is useful to look at their implementation as well. Listing 18 depicts the implementation of Method `equal`. The method first assigns value `v` to variable `x` and then call the constraint propagation on the solver. This call is the connection between the search and propagation. Note also that, if the constraint propagation fails, an exception is caught and the state is restored for the next branch (lines 18–24 in Listing 17).

It is worthwhile mentioning one difference between Algorithm 3 and the implementation in Listing 17. Line 7 of Algorithm 3 is implemented by line 13 in Listing 17. But the implementation does not manipulate constraints directly. Rather it receives an array of closures that, when called, will apply these constraints.

5.2 State Specification

The state in MiniCP is represented by classes implementing two interfaces, `StateInt` and `StateBool`, that encapsulate an integer and a Boolean respectively. These state variables are created by the state manager. For conciseness only `StateInt` implementation is discussed next. Listing 19 depicts the `StateInt` interface which supports setting a new value and accessing the current value. Listing 20 depicts its use and the intended behavior assuming an existing `StateManager sm`. The implementation of state managers exploits the fact MiniCP uses a LIFO strategy in its search procedures, which is obviously the case for depth-first search.

Copying State Management The simplest state-management strategy consists in creating a backup of the state, saving the values of all the `StateInt` and `StateBool` instances for future restoration. Listing 21 illustrates this implementation. Lines 1–3 show the interface of a backup entry, which has only one capability: the ability to restore its previous value. The main class, `Copier`, implements the `StateManager` interface. Its internal state consists of two containers. First, variable `store` keeps track of all state objects ever created. Factory methods `makeStateInt` and `makeStateBool` adds a reference to any object they create to this container, as shown in lines 34–38. Second, variable `prior` contains a stack of states which are saved by method `saveState` and restored by method `restoreState`. Method `saveState` creates an instance of the nested class `Backup` whose constructor saves a copy of every object in the state store. Method `restore` of this backup object restores the saved values. Note that the backup object also saves the size of store since state objects may be created during the search and the proper size must be restored. The state objects are created by methods `makeStateInt` and `makeStateBool` which returns objects of type `CopyInt` and `CopyBool`. The code for `CopyInt` is shown in Listing 22. Its `save` method creates a backup entry which is an instance of the nested class `CopyIntStateEntry`. Method `restore` of the nested class exploits the fact that it can refer to the `this` object of the encapsulating class.

Listing 21: The Copier.

```
public interface StateEntry {
    public void restore();
}

public class Copier implements StateManager {
    class Backup extends Stack<StateEntry> {
        private int sz;
        Backup() {
            sz = store.size();
            for (Storage s : store)
                add(s.save());
        }
        void restore() {
            store.setSize(sz);
            for (StateEntry se : this)
                se.restore();
        }
    }
    private Stack<Storage> store;
    private Stack<Backup> prior;
    public Copier() {
        store = new Stack<Storage>();
        prior = new Stack<Backup>();
    }
    public int getLevel()                { return prior.size() - 1;}
    @Override public void saveState()    { prior.add(new Backup());}
    @Override public void restoreState() { prior.pop().restore();}
    @Override public void withNewState(Procedure body) {
        final int level = getLevel();
        saveState();
        body.call();
        while (getLevel() > level) restoreState();
    }
    @Override public StateInt makeStateInt(int initValue) {
        CopyInt s = new CopyInt(initValue);
        store.add(s);
        return s;
    }
    @Override public StateBool makeStateBool(boolean initValue) {
        CopyBool s = new CopyBool(initValue);
        store.add(s);
        return s;
    }
}
```

Listing 22: The CopyInt Representation.

```
public class CopyInt implements Storage, StateInt {
    class CopyIntStateEntry implements StateEntry {
        private final int v;
        public CopyIntStateEntry(int v) { this.v = v;}
        @Override public void restore() { CopyInt.this.v = v;}
    }
    private int v;
    protected CopyInt(int initial)       { v = initial;}
    @Override public int setValue(int v) { this.v = v;return v;}
    @Override public int value()         { return v;}
    @Override public String toString()   { return String.valueOf(v);}
    @Override public StateEntry save()   { return new CopyIntStateEntry(v);}
}
```

Trailing State Management Saving *all* state variables is typically wasteful as only a few state variables are modified during a propagation step. MINICP, like most constraint-programming implementations, uses a technique called *trailing* to lazily copy the state. In other words, trailing only copies the variables

Draft September 15, 2018

Listing 23: Trailer

```
public class Trailer implements StateManager {
    class Backup extends Stack<StateEntry> {
        void restore() {
            for (StateEntry se : this)
                se.restore();
        }
    }
    private Stack<Backup> prior;
    private Backup current;
    private long magic = 0L;

    public Trailer() {
        prior = new Stack<Backup>();
        current = new Backup();
    }
    public long getMagic() { return magic;}
    public void pushState(StateEntry entry) { current.push(entry);}
    @Override public int getLevel()         { return prior.size() - 1;}
    @Override public void saveState() {
        prior.add(current);
        current = new Backup();
        magic++;
    }
    @Override public void restoreState() {
        current.restore();
        current = prior.pop();
        magic++;
    }
    @Override public void withNewState(Procedure body) {
        final int level = getLevel();
        saveState();
        body.call();
        while (getLevel() > level) restoreState();
    }
    @Override public StateInt makeStateInt(int initValue) {
        return new TrailInt(this,initValue);
    }
    @Override public StateBool makeStateBool(boolean initValue) {
        return new TrailBool(this,initValue);
    }
}
```

that are actually modified. The trailing mechanism described in this section was first implemented in constraint programming by the CHIP system [24,1]. Knuth [11] attributes the first formulation of trailing and its usage in a backtracking algorithm to Floyd [6].

Listings 23 presents an implementation of the trail. Saving a state (method `saveState`) simply pushes the current backup on top of the backup stack (`prior`) (in constant time) and creates a new *empty* backup for the next batch of changes. Similarly, restoring the state (method `restoreState`) instructs the current backup to restore all changes that were tracked since the last call to `saveState` and then reinstates the previous backup from the `prior` stack. Lazily saving changes made to the state is the responsibility of the state objects. Those are created from the factory methods `makeStateInt` and `makeStateBool` which return, respectively, an instance of `TrailInt` or `TrailBool`.

Consider the implementation of `TrailInt` shown in Listing 24. The big difference with the `Copier` is that now the saving is lazy as it only happens when modification is performed an object and untouched objects are never backed up. The `TrailInt` class implements the `StateInt` interface and is a cousin of `CopyInt`. Its mutator method `setValue` is at the core of the implementation. If the new value differs from the current value, it first *trails* (i.e., it backs up) the old value on the trail with an instance of the nested class `StateEntryInt` and then modifies its attribute v.

The implementation features a simple optimization that avoids trailing an object again if it has already been saved in the current backup. This is done by using an integer identifying the current backup (the `magic` attribute of the trail) and saving, inside the `TrailInt` instance, the `magic` value when the object is trailed. The `magic` attribute is incremented every time a backup is saved and restored.

Draft September 15, 2018

Listing 24: Trailable Integer.

```
1  public class TrailInt implements StateInt {
2      class StateEntryInt implements StateEntry {
3          private final int v;
4          public StateEntryInt(int v)      { this.v = v;}
5          @Override public void restore() { TrailInt.this.v = v;}
6      }
7      private Trailer trail;
8      private int v;
9      private long lastMagic = -1L;
10     protected TrailInt(Trailer trail, int initial) {
11         this.trail = trail;
12         v = initial;
13     }
14     private void trail() {
15         long trailMagic = trail.getMagic();
16         if (lastMagic != trailMagic) {
17             lastMagic = trailMagic;
18             trail.pushState(new StateEntryInt(v));
19         }
20     }
21     @Override public int setValue(int v) {
22         if (v != this.v) {
23             trail();
24             this.v = v;
25         }
26         return this.v;
27     }
28     @Override public int value()       { return this.v;}
29     @Override public String toString() { return "" + v;}
30 }
```

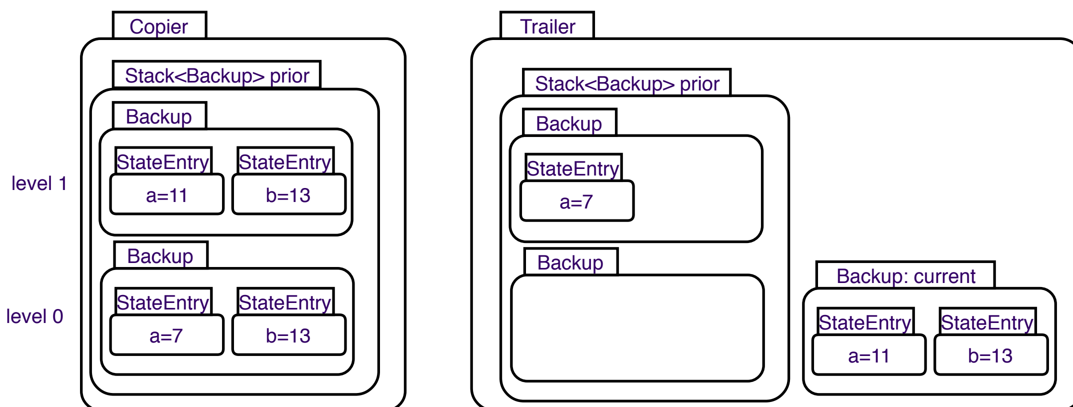

Fig. 3: Illustration of the differences between Copier and Trailer related to example given in Listing 20.

Figure 3 contrasts the two state management policies implemented by the `Copier` (left) and `Trailer` (right) in MiniCP for the example shown in Listing 20. It is worth noting that the `Trailer` avoids creating unnecessary copies for untouched state objects.

5.3 Revisiting the Domain Implementation

The domain of a variable is part of the state and must be saved and restored. So it is necessary to upgrade the implementation presented previously. The sparse set representation is convenient for this purpose. Assume that method `saveState` is called when the domain has size 8. In all subsequent forward computations, the array will contain a permutation of these 8 values in the first 8 slots. So, when restoring the state, it suffices to restore the size of the domain. Consider Figure 3 again and assume that the state was saved before the removal of values 4 and 6. Then a `restoreState` operation restores the `size` to 9 (as it was at the time of the `saveState`) and the two removed values are reinserted in the set

Draft September 15, 2018

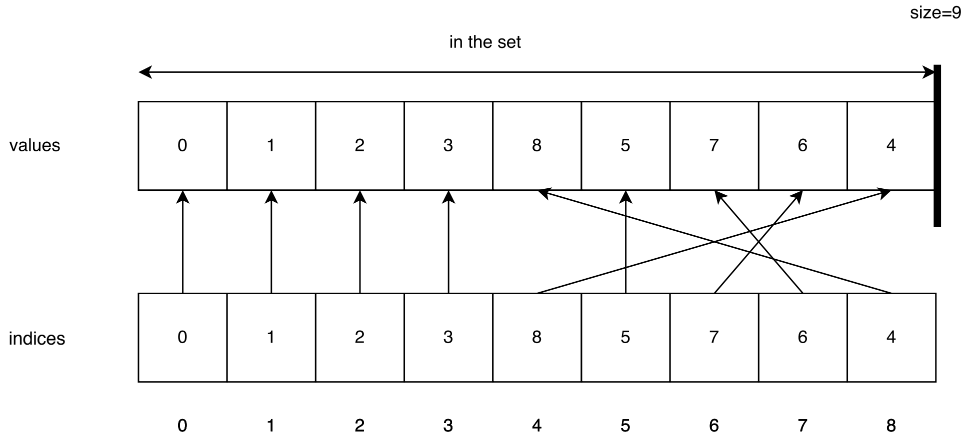

Fig. 4: Sparset Set

Listing 25: The StateStack Implementation

```
1 public class StateStack<E> {
2     StateInt      size;
3     ArrayList<E> stack;
4     public StateStack(StateManager sm) {
5         size = sm.makeStateInt(0);
6         stack = new ArrayList<E>();
7     }
8     public void push(E elem) {
9         stack.add(size.value(),elem);
10        size.increment();
11    }
12    public int size()        { return size.value(); }
13    public E get(int index) { return stack.get(index); }
14 }
```

as expected. The permutation of values is not the same as when `saveState` was called but the domain captures the same set of values $\{0, \ldots, 8\}$. This is depicted in Figure 4. For this reason, MiniCP simply provides a `StateSparsetSet` class which is the same as `SparsetSet` except that instance variable `size` is a state variable.

Finally, observe that the sets of constraints stored in variables are also part of the state: Constraints can be added (and backtracked over). These sets can be implemented with a `StateStack` data structure whose implementation can simply use a `StateInt` to represent the size of the stack. On backtracking, the correct size, and hence the correct set of constraints, are restored. The brief implementation is shown in Listing 25.

5.4 Supporting Optimization

The resolution of a Constraint Optimization Problem (COP) reduces to solving a sequence of constraint satisfaction problem. Without loss of generality, assume that the COP is a minimization. The basic idea behind a branch and bound is the following. Given a COP $\langle X, \mathcal{D}, C, f\rangle$, first solve the CSP $\langle X, \mathcal{D}, C\rangle$. As soon as a solution σ is found, evaluate $f(\sigma)$ to obtain a first primal bound f_1 on the objective function and solve the second CSP $\langle X, \mathcal{D}, C \cup \{f < f_1\}\rangle$. Repeat this process until the kth CSP $\langle X, \mathcal{D}, C \cup \{f < f_k\}\rangle$ becomes infeasible. This failure proves the optimality of the last feasible solution.

The implementation of MiniCP follows this scheme but with an important difference: It does not restart the search from scratch. Instead, it continuously tightens the objective $f < b$, where b represents the value of the best-found solution at some computation stage. The implementation is given in Listings 26 and 27 which show how to minimize the value of a variable. Listing 26 presents a class `Minimize` that implements interface `Objective` and supports only one method `tighten` which is called when the solver finds a solution (see line 16 in Listing 27). Its implementation simply tightens the best primal bound (i.e., the value of the best solution found so far) before failing to search for better solutions

Draft September 15, 2018

Listing 26: Minimization Objective

```
1 public class Minimize implements Objective {
2     public int bound = Integer.MAX_VALUE;
3     private final IntVar x;
4     public Minimize(IntVar x) {
5         this.x = x;
6         x.getSolver().onFixPoint(() -> x.removeAbove(bound));
7     }
8     public void tighten() {
9         if (!x.isBound()) throw new RuntimeException("objective not bound");
10        this.bound = x.max() - 1;
11        throw new InconsistencyException();
12    }
13 }
```

Listing 27: Refinement for `DFS`.

```
1 public class DFS {
2    private StateManager sm;
3    private Supplier<Procedure[]> branchingScheme;
4    private List<Procedure>       solObservers = new LinkedList<Procedure>();
5    public DFS(StateManager sm,Supplier<Procedure[]> b) {
6       this.sm = sm;branchingScheme = b;
7    }
8    public void onSolution(Procedure p)  { solObservers.add(p);}
9    public void notifySolution()         { solObservers.forEach(p -> p.call());}
10   public void solve() {
11      sm.withNewState( () -> {
12        dfs();
13      });
14   }
15   public void optimize(Objective obj) {
16      onSolution(() -> obj.tighten());
17      sm.withNewState( () -> {
18        dfs();
19      });
20   }
21   // as before....
22 }
```

(lines 8–11). The implementation uses observers, i.e., closures attached to the search and executed when a solution is found.

It remains to ensure that the variable being minimized has the proper bound at all times. The bound update cannot be performed only when a new solution is found, since the update would be undone on backtracking. MiniCP applies the bound update systematically before each fixpoint (and hence after each branching), using an observer again (line 6 in Listing 26 adds the bound update to the fixpoint observer).

The extended `DFS` class in Listing 27 adds an `optimize` method to complement its `solve` method. The `onSolution` method registers listener closures in a `solObserver` list. The `notifySolution` method simply calls all the closures registered in the solution listener. Finally, the `optimize` method is modeled after `solve` –repeated above for clarity– and simply registers a solution listener that tightens the primal bound of the objective when a solution is produced. A similar implementation in the `MiniCP` class effectively implements the `onFixPoint` observer which is triggered as the first step of the `fixpoint` method of the solver.

6 Advanced Filtering Techniques

This section introduces some more advanced features of MiniCP for constraint propagation. In particular, this section presents variable views, anonymous constraints, reified constraints, and global constraints.

Draft September 15, 2018

Listing 28: N-Queens model with Views

```
1 for(int i=0;i < n;i++)
2   for(int j=i+1;j < n;j++) {
3     cp.post(notEqual(q[i],q[j]));
4     cp.post(notEqual(plus(q[i],i),plus(q[j],j)));
5     cp.post(notEqual(minus(q[i],i),minus(q[j],j)));
6   }
```

Draft September 15, 2018

6.1 Variable Views

The N-Queens program of Listing 1 states constraints of the form

$$q_i + i \neq q_j + j \wedge q_i - i \neq q_j - j$$

by using a ternary constraint of signature

```
public NotEqual(IntVar x, IntVar y, int c)
```

which holds if $x \neq y + c$. The approach of creating many variants of a constraint is motivated by efficiency considerations. Indeed, it is possible to rewrite each of the above constraints in terms of binary disequations, e.g.,

$$qp_i = q_i + i \wedge qp_j = q_j + j \wedge qp_i \neq qp_j,$$

by introducing many intermediate variables and constraints. This removes the need for constraint variants but may slow down the overall efficiency of the fixpoint algorithm significantly. Fortunately, there is a third alternative that strikes a good compromise between efficiency and simplicity: variable views [22].[2] A variable view is an `IntVar` variable that represents an affine transformation over a given variable. Views in MiniCP are built through the following classes

- `IntVarViewMul` for $c * X$,
- `IntVarViewOffset` for $X + o$ and
- `IntVarViewOpposite` for $-X$.

which can be composed to obtain affine transformations. The queens model using views is shown in Listing 28. Methods `plus` and `minus` creates views of type `IntVarViewOffset`, which are then used in binary disequations.

The implementation of views is illustrated in Listing 29 using class `IntVarViewOffset`. Queries on the views are delegated to the view variable after possibly adding or subtracting the offset. Value removal also delegates to the view variable after having subtracted the offset. Finally, constraints on the views are attached to the view variable.

6.2 Anonymous Constraints

Sometimes the `propagate` method of a constraint is so small that it is cumbersome to build an entire class. MiniCP provides anonymous constraints to address these cases. Variables provide methods such as `whenBind` and `whenDomainChange` that receive as input a closure, create constraint whose `propagate` method calls the closure, and link the resulting constraints to the proper list in the variables.

Listing 30 illustrates anonymous constraints for the implementation of the `NotEqual` constraint. The constraint has no `propagate` method, all the constraints being created in method `post`. In particular, lines 15–16 specify what happens when variables `x` or `y` are bound. Anonymous constraints are particularly appealing in the sense that they can capture part of the lexical scope, providing compact implementations of many small constraints.

[2] Expressions are a fourth, more general, alternative which was implemented in CHIP, Ilog Solver, and many subsequent solvers; They require a heavier machinery.

Listing 29: The Offset View

```
1 public class IntVarViewOffset implements IntVar {
2     private final IntVar x;
3     private final int o;
4     public IntVarViewOffset(IntVar x, int offset) { // y = x + o
5         this.x = x;
6         this.o = offset;
7     }
8     public Solver getSolver()        { return x.getSolver();}
9     public int min()                 { return x.min() + o;}
10    public int max()                 { return x.max() + o;}
11    public int size()                { return x.size();}
12    public boolean contains(int v)   { return x.contains(v - o);}
13    public boolean isBound()         { return x.isBound();}
14
15    public void remove(int v)        { x.remove(v - o);}
16    public void assign(int v)        { x.assign(v - o);}
17    public void removeBelow(int v)   { x.removeBelow(v - o);}
18    public void removeAbove(int v)   { x.removeAbove(v - o);}
19
20    public void propagateOnDomainChange(Constraint c){ x.propagateOnDomainChange(c);}
21    public void propagateOnBind(Constraint c)        { x.propagateOnBind(c);}
22    public void propagateOnBounds(Constraint c)      { x.propagateOnBounds(c);}
23 }
```

Listing 30: NotEqual Constraint

```
1 public class NotEqual extends AbstractConstraint {
2     private final IntVar x, y;
3     private final int c;
4     public NotEqual(IntVar x, IntVar y, int c) { // x != y + c
5         super(x.getSolver());
6         this.x = x;this.y = y;this.c = c;
7     }
8     public NotEqual(IntVar x, IntVar y) { this(x,y,0);}
9     @Override public void post()  {
10        if (y.isBound())
11            x.remove(y.min() + c);
12        else if (x.isBound())
13            y.remove(x.min() - c);
14        else {
15            x.whenBind(() -> y.remove(x.min() - c));
16            y.whenBind(() -> x.remove(y.min() + c));
17        }
18    }
19  }
```

6.3 Reified Constraints

The ability of reasoning about constraints in constraint-programming systems was introduced in cc(FD) [26,30]. It is available in most modern systems through reified constraints (also called indicator constraints in mathematical programming). A reified constraint links a Boolean variable with the truth value of a constraint (0 is false and 1 is true). For instance, the reified constraint `isEqual(x,v,b)` holds if $b \equiv (x = v)$. Its implementation captures the following inference rules:

when $\mathcal{D}(b) = 1$ then add constraint $x = v$;
when $\mathcal{D}(b) = 0$ then add constraint $x \neq v$;
when $\mathcal{D}(x) = v$ then add constraint $b = 1$;
when $v \notin \mathcal{D}(x)$ then add constraint $b = 0$.

The implementation is given in Listing 31 and makes heavy use of anonymous constraints.

Draft September 15, 2018

Listing 31: IsEqual

```
1  public class IsEqual extends Constraint { // b <=> x == v
2      private final BoolVar b;
3      private final IntVar x;
4      private final int v;
5      public IsEqual(BoolVar b, IntVar x, int v) {
6          super(x.getSolver());
7          this.b = b;
8          this.x = x;
9          this.v = v;
10     }
11     public void post() throws InconsistencyException {
12         if (b.isTrue())
13             x.assign(v);
14         else if (b.isFalse())
15             x.remove(v);
16         else if (x.isBound())
17             b.assign(x.min() == v);
18         else if (!x.contains(v))
19             b.assign(0);
20         else {
21             b.whenBind(() -> {
22                 if (b.isTrue())
23                    x.assign(v);
24                 else
25                    x.remove(v);
26             });
27             x.whenBind(() -> b.assign(x.min() == v));
28             x.whenDomainChange(() -> {
29                 if (!x.contains(v))
30                     b.assign(0);
31             });
32         }
33     }
34 }
```

Draft September 15, 2018

6.4 Global Constraints

Global constraints are a key aspect of constraint programming: They capture combinatorial substructures that are present in many application and which enjoy fast filtering algorithms. This section presents three simple examples of global constraints which are used in the QAP model given in Listing 3: 1) `sum(weightedDist)`, 2) `element(d,x[i],x[j])` and 3) `allDifferent(x)`. Readers interested in global constraints should consult the global constraint catalog [2] or the surveys [8,9].

The Sum Constraint The `sum` constraint enforces the relation $0 = \sum_{i=0}^{n-1} x_i$. Enforcing domain consistency for `sum` is NP-hard and this section presents an algorithm enforcing bound consistency. The bound-consistent inference rules for `sum` are:

- $\max(\mathcal{D}(x_i)) \leftarrow \sum_{j \neq i} - \min(\mathcal{D}(x_j))$
- $\min(\mathcal{D}(x_i)) \leftarrow \sum_{j \neq i} - \max(\mathcal{D}(x_j))$

Computing those rules for every variable takes $\mathcal{O}(n)$ per variable and hence $\mathcal{O}(n^2)$ overall. The complexity can be reduced to $\mathcal{O}(n)$ by precomputing $\text{sumMax} = \sum_j \max(\mathcal{D}(x_j))$ and $\text{sumMin} = \sum_j \min(\mathcal{D}(x_j))$. Those precomputed values allow for a $\mathcal{O}(1)$ inference rule for each variable:

1. $\max(\mathcal{D}(x_i)) \leftarrow \min(\mathcal{D}(x_i)) - \text{sumMin}$
2. $\min(\mathcal{D}(x_i)) \leftarrow \max(\mathcal{D}(x_i)) - \text{sumMax}$

In practice, further efficiency can be obtained by exploiting the incremental nature of the filtering process. Indeed, the number of bound variables when going down a branch of the depth-first search can never decrease. Hence, it is possible to pre-compute incrementally the sum of these bound variables and to iterate only on free variables. The implementation uses an integer array `free` and a counter nf to represent the indices of the variables and the number of free variables with the following invariants: The first nf indices in the array `free` represent free variables while the remaining

variables are bound. The implementation also maintains sumBounds, the partial sum of bound variables sumBounds $= \sum_{i \geq \text{nf}} x_{\text{free}[i]}$. The values sumMax and sumMin can then be computed in $\mathcal{O}(\text{nf})$ as follows: sumMax = sumBounds + $\sum_{j<\text{nf}} \max(\mathcal{D}(x_{free[j]})$ and sumMax = sumBounds + $\sum_{j<\text{nf}} \min(\mathcal{D}(x_{free[j]})$. Only the free variables are considered for filtering, i.e., $\forall i < \text{nf}$:

1. $\max(\mathcal{D}(x_{free[j]})) \leftarrow \min(\mathcal{D}(x_{free[j]})) - \text{sumMin}$
2. $\min(\mathcal{D}(x_{free[j]})) \leftarrow \max(\mathcal{D}(x_{free[j]})) - \text{sumMax}$

The complete code in Listing 32. The time complexity for one call to `propagate` is $\Theta(\text{nf})$. The invariant on array free is maintained by iterating over the free variables from index $\text{nf} - 1$ until 0 and in swapping, in $\mathcal{O}(1)$ time, any index whose variable is bound with the last free variable at index nf. The value assigned to the bound variable is also added to sumBounds. The set of bound variables only increases monotonically in a branch: Hence storing nf as a `StateInt` ensures that the invariant is still valid when backtracking. Obviously, this is similar to the state representation of sparse sets.

The Element Constraint The *element* constraint offers the ability of indexing arrays with decision variables [24,25]. This functionality often avoids the introduction of many binay variables, as well as arrays of binary variables with many dimensions. This section considers an element constraint of the form $z = d_{x,y}$, where d is a two-dimensional array of constants of size $n \times m$ and x, y, and z are variables. Moreover, the proposed implementation achieves domain consistency for variables x and y and bound consistency for variable z. This corresponds to a frequent case in practice; The implementation can easily be generalized to enforce domain consistency on all variables.

The implementation first builds a sorted set of tuples $\langle i_1, j_1, v_1 \rangle, \ldots, \langle i_p, j_p, v_p \rangle$ where $v_1 \leq \ldots \leq v_p$, $p = nm$, $0 \leq i_k < n$, $0 \leq j_k < m$, and $v_k = d_{i_k, j_j}$. This sorted set can be computed once in method `post` and makes it simple to filter variable z. The implementation scans the sorted set from below until it finds a tuple $\langle i_m, j_m, v_m \rangle$ such that $i_m \in \mathcal{D}(x)$, $j_m \in \mathcal{D}(y)$, and $v \geq \min(\mathcal{D}(z))$, producing v as the new lower bound. The new upper bound can be obtained similarly by iterating from above.

The filtering of variables x and y is slightly more involved. Consider variable x (variable y is similar). A value i for x may appear in many tuples and it is only when all these tuples cannot be solutions that value i can be removed from $\mathcal{D}(x)$. To implement the filtering efficiently, the implementation uses a counter $c_{x,i}$ (a `StateInt` variable) that represents the number of times value i appears in a possible solution, i.e., a tuple $\langle i, j, v \rangle$ such that $i_m \in \mathcal{D}(x)$, $j_m \in \mathcal{D}(y)$, and $\min(\mathcal{D}(z)) \leq v \leq \max(\mathcal{D}(z))$. These counters are assigned to m initially, since no processing takes place in method `post`.

Method `propogate` filters all variables in two scans: one from below and one from above. When iterating from below (the other case is similar), the implementation considers each tuple. If the tuple $\langle i, j, v \rangle$ is a possible solution, the sweep stops and the minimum of variable z is updated to v. Otherwise, the counters $c_{x,i}$ and $c_{y,j}$ are decremented. If a counter reaches zero, then the value is removed from the domain of the corresponding variable. To avoid considering a tuple more than once, the implementation also maintains two indices `low` and `up` (represented by `StateInt`) to remember where to start the below and above scan of the sorted set of tuples. The full implementation is given in Listing 33.

The AllDifferent Constraint The `allDifferent`$(x_0, \ldots, x_{n-1})$ constraint ensures that every variable in $\{x_0, \ldots, x_{n-1}\}$ takes a different value. Decomposition into elementary binary constraints such as $\forall i \neq j : x_i \neq x_j$ does not enforce domain consistency globally. For instance, the decomposition does not detect inconsistency for the domains $\mathcal{D}(x_0) = \mathcal{D}(x_1) = \mathcal{D}(x_2) = \{1, 2\}$, which is directly implied by the pigeonhole principle. It is interesting to outline how to implement consistency on the global constraint to highlight how constraint programming may leverage combinatorial algorithms.

Detecting feasibility can be achieved by solving a maximum matching problem in the bipartite value graph $G(V_1, V_2, E)$ with $V_1 = \{x_0, \ldots, x_{n-1}\}$ and $V_2 = \bigcup_{i \in 0..n-1} = \mathcal{D}(x_i)$, $E = \{(i, v) \mid v \in \mathcal{D}(x_i)\}$. The constraint is satisfiable if and only if the maximum matching has cardinality n. The rest of the presentation assumes that $V_2 = \{0, \ldots, m-1\}$ without loss of generality.

Filtering the `allDifferent` constraint amounts to determining whether an edge in the bipartite graph belongs to some maximum matching. This can be achieved by solving a matching problem for every edge. Régin [?] however showed that this filtering can be performed in time linear in the size of the graph. The algorithms is composed of four steps illustrated in Figure 5 and summarized as follows:

1. A maximum-size matching M is computed in the value graph.
2. The residual graph is constructed.
3. The strongly connected components (SCC) are computed in the residual graph.

Draft September 15, 2018

Listing 32: Sum

```java
public class Sum extends AbstractConstraint {
    private  int[] free;
    private StateInt nFrees;
    private StateInt sumBounds;
    private IntVar [] x;
    private int n;

    public Sum(IntVar [] x) {
        super(x[0].getSolver());
        this.x = x;
        this.n = x.length;
        nFrees = cp.getStateManager().makeStateInt(n);
        sumBounds = cp.getStateManager().makeStateInt(0);
        free = new int[n];
        for (int i = 0; i < n; i++)
            free[i] = i;
    }

    public void post() throws InconsistencyException {
        for (IntVar var: x)
            var.propagateOnBoundChange(this);
        propagate();
    }
    public void propagate() throws InconsistencyException {
        // update partial sum and maintain invariant
        int nf = nFrees.getValue();
        int partialSum = sumBounds.getValue();
        for (int i = nf - 1; i >= 0; i--) {
            int idx = free[i];
            IntVar y = x[idx];
            if (y.isBound()) {
                partialSum += y.min(); // Update partial sum
                // Swap the variable
                free[i] = free[nf - 1];
                free[nf - 1] = idx;
                nf--;
            }
        }
        sumBounds.setValue(partialSum);
        nFrees.setValue(nf);
        int sumMax = partialSum;
        int sumMin = partialSum;
        for (int i = nf - 1; i >= 0; i--) {
            int idx = free[i];
            sumMax += x[idx].max();
            sumMin += x[idx].min();
        }
        // filter free variables
        if (sumMin > 0 || sumMax < 0)
            throw new InconsistencyException();
        for (int i = nf - 1; i >= 0; i--) {
            int idx = free[i];
            x[idx].removeAbove(-(sumMin-x[idx].min()));
            x[idx].removeBelow(-(sumMax-x[idx].max()));
        }
    }
}
```

4. Every edge belonging to some SCC belongs to a maximum matching and edges between SCCs do not. Hence these edges can be filtered.

A sketch of the implementation is given in Listing 34. The `MaximumMatching` object is in charge of computing the maximum matching in the value graph. This object wraps an augmenting path algorithm. Although the details of the implementation are not given here, this algorithm is incremental in the following sense: From one call to the next at line 29, it restarts its computation from the previous matching and needs only as many augmenting steps as the number of deleted edges. In addition, the matching remains valid upon backtracking. The feasibility test is computed at line 30, throwing

Draft September 15, 2018

Listing 33: Element 2D

```
1 public class Element2D extends AbstractConstraint {
2    private final IntVar x, y, z;
3    private final int n, m;
4    private final StateInt[] nRowsSup;
5    private final StateInt[] nColsSup;
6    private final StateInt low,up;
7    private final ArrayList<Triple> xyz;
8    private class Triple implements Comparable<Triple> {
9       protected final int x,y,z;
10      private Triple(int x, int y, int z) { this.x = x;this.y = y;this.z = z;}
11      @Override public int compareTo(Triple t) { return z - t.z;}
12   }
13   public Element2D(int[][] d, IntVar x, IntVar y, IntVar z) {
14      super(x.getSolver());
15      this.x = x;
16      this.y = y;
17      this.z = z;
18      n = d.length;
19      m = d[0].length;
20      xyz = new ArrayList<Triple>();
21      for (int i = 0; i < d.length; i++)
22          for (int j = 0; j < d[i].length; j++)
23              xyz.add(new Triple(i,j,d[i][j]));
24      Collections.sort(xyz);
25      StateManager sm = cp.getStateManager();
26      low = sm.makeStateInt(0);
27      up  = sm.makeStateInt(xyz.size()-1);
28      nColsSup = IntStream.range(0,n).mapToObj(i->sm.makeStateInt(m)).toArray(StateInt[]::
          new);
29      nRowsSup = IntStream.range(0,m).mapToObj(i->sm.makeStateInt(n)).toArray(StateInt[]::
          new);
30   }
31   @Override public void post()  {
32      x.removeBelow(0);
33      x.removeAbove(n-1);
34      y.removeBelow(0);
35      y.removeAbove(m-1);
36      x.propagateOnDomainChange(this);
37      y.propagateOnDomainChange(this);
38      z.propagateOnBoundChange(this);
39      propagate();
40   }
41   private void updateSupports(int lostPos)  {
42      if (nColsSup[xyz.get(lostPos).x].decrement() == 0) x.remove(xyz.get(lostPos).x);
43      if (nRowsSup[xyz.get(lostPos).y].decrement() == 0) y.remove(xyz.get(lostPos).y);
44   }
45   @Override public void propagate() {
46      int l    = low.getValue(),u = up.getValue();
47      int zMin = z.min(),zMax  = z.max();
48      while(xyz.get(l).z < zMin|| !x.contains(xyz.get(l).x)|| !y.contains(xyz.get(l).y)){
49         updateSupports(l++);
50         if (l > u) throw new InconsistencyException();
51      }
52      while(xyz.get(u).z > zMax|| !x.contains(xyz.get(u).x)|| !y.contains(xyz.get(u).y)){
53         updateSupports(u--);
54         if (l > u) throw new InconsistencyException();
55      }
56      z.removeBelow(xyz.get(l).z);
57      z.removeAbove(xyz.get(u).z);
58      low.setValue(l);
59      up.setValue(u);
60   }
61 }
```

an `InconsitencyException` in case $|M| < n$. Line 32 updates the residual graph. Line 33 computes the SCC. e.g, using Tarjan's or Kosaraju's algorithm. This computation is hidden in the method `GraphUtil.stronglyConnectedComponents(g)` where `g` is the residual graph. It returns an array

Draft September 15, 2018

Listing 34: AllDifferent

```
1 public class AllDifferent extends AbstractConstraint {
2    private IntVar[] x;
3    private final int nVar, maxVal;
4    private int[] match; // for each var the value it is matched to
5    private final MaximumMatching maximumMatching;
6    // residual graph
7    private Graph g = new Graph() { /* ... omitted ... */ };
8
9    public AllDifferent(IntVar... x) {
10      super(x[0].getSolver());
11      maximumMatching = new MaximumMatching(x);
12      match = new int[x.length];
13      this.x = x;
14      this.nVar = x.length;
15   }
16   @Override public void post()  {
17       for (int i = 0; i < nVar; i++)
18          x[i].propagateOnDomainChange(this);
19       propagate();
20   }
21
22   // update the range of values minVal,maxVal
23   public void updateRange()  { /* ... omitted ... */ }
24
25   // update adjacency lists of the residual graph
26   public void updateGraph() {/* ... omitted ... */ }
27
28   @Override public void propagate() {
29      int size = maximumMatching.compute(match); // step1
30      if (size < nVar)
31           throw InconsistencyException.INCONSISTENCY;
32      updateRange(); updateGraph(); // step2
33      int[] scc = GraphUtil.stronglyConnectedComponents(g); // step3
34      for (int i = 0; i < nVar; i++) {
35        for (int v = 0; v <= maxVal; v++)
36           if (match[i] != v && scc[i] != scc[v + nVar])
37              x[i].remove(v); // step4
38  }
39 }
```

associating a SCC identifier with each node of the graph (i.e., each variable and value). Finally, line 37 removes any value v from the domain of a variable x if the corresponding edge is not in M and if the SCCs of x and v are different. A comprehensive discussion of this algorithm can be found in [**?**].

7 Advanced Search Techniques

This section presents some advanced search techniques. It shows how to implement search limits, combinators, and large neighborhood search in MiniCP.

7.1 Search Limits

In various circumstances, it is desirable to terminate the search before having found all solutions or the optimal solution. MiniCP provides a simple abstraction to implement various search limits, e.g., on the number of solutions, the CPU time, or the number of failures. Method `solve` accepts a *predicate* (a Java closure) which is applied at every node of the search to decide whether to terminate the search early. For instance, the fragment

```
solver.solve(stats -> stats.nSolutions == 1);
```

illustrates how to terminate the search once the first solution has been found. The closure receives an argument `stat` (of type `SearchStatistics`) that collects statistics about the search. In this example, the boolean predicate simply checks the number of solutions recorded in *stat*. The generalized

Draft September 15, 2018

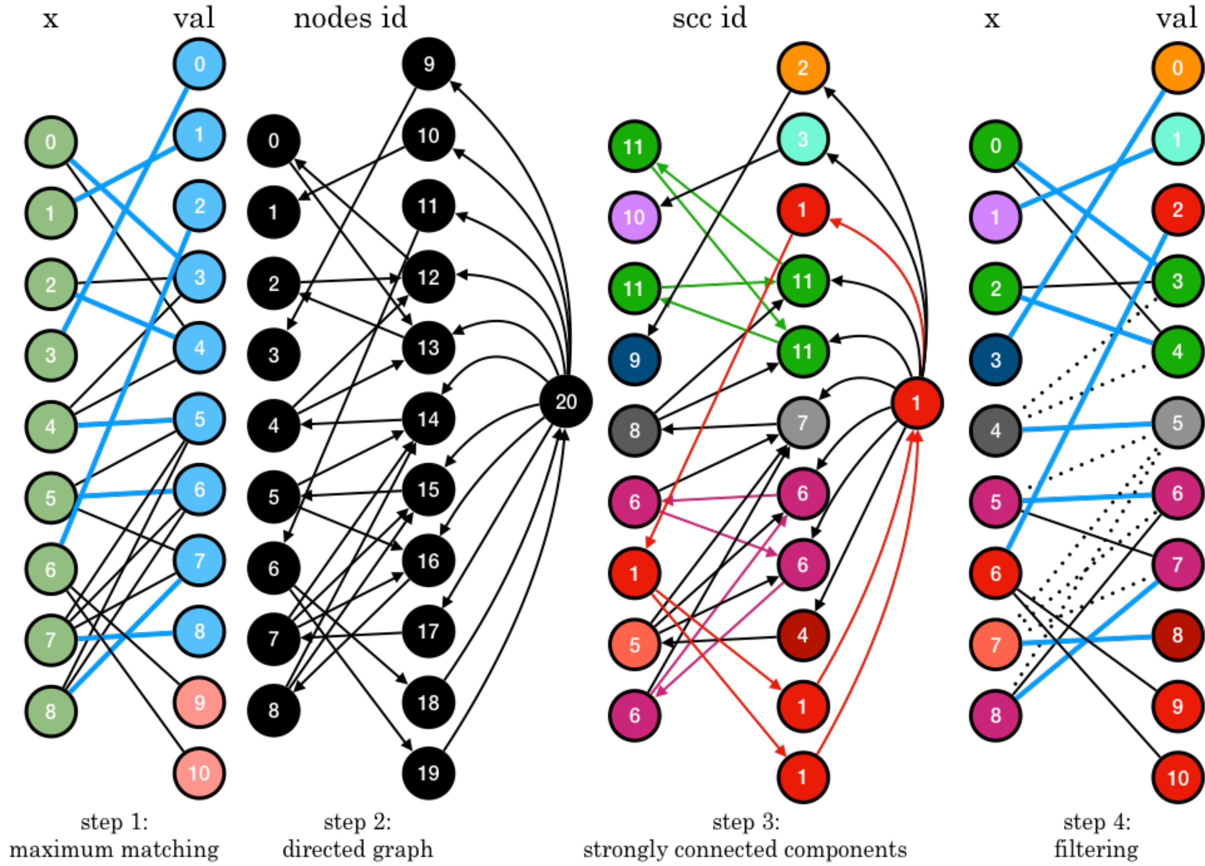

Fig. 5: Illustration of the `allDifferent` constraint. Step 1: nodes contain the variable indices and values. Step 2: nodes contain canonical id's in the residual graph. Step 3: component ids are given in each node and colors represent those ids. Step 4: removed edges are represented with dashed lines.

Draft September 15, 2018

implementation of depth-first search is shown in Listing 35. Line 8 contains the more general `solve` call. The DFS search in line 16 receives, as inputs, the statistics object and the stopping predicate, which are used before branching to determine whether to terminate the search.

7.2 Search Combinators

Search combinators in MiniCP compose branching schemes to produce more sophisticated branching schemes. This section illustrates how to build search combinators in MiniCP through an example.[3] Listing 36 shows how to implement search phases, where the search composes several branching schemes in sequence. Such a combinator is useful to implement search phases which are often used to prioritize the assignment of certain variables over others.

Recall that a branching scheme returns an array of branches, which is empty when the scheme completes. To compose a sequence of branching schemes, it suffices, at each node of search tree, to iterate over the schemes until one produces a non-empty array of branches. The `sequencer` combinator receives, as input, an array of branching schemes. When called to produce a set of branches, it iterates over the branching schemes until one produces a non-empty array (lines 7–11). Note the compositional nature of the combinator: The combinator itself is a branching scheme and it applies to any branching scheme.

Combinators can be composed with generic selectors to specify complex branching schemes concisely. Consider the implementation of a search procedure that assigns two arrays of variables `x` and `y` in sequence, using the first-fail principle (i.e., the variable with the smallest domain is assigned first). The high-level specification is as follows:

[3] For more discussion on each combinators, readers can consult [16, 28, 20].

Listing 35: Core DFS Skeleton

```
1 public class DFS {
2    private StateManager sm;
3    private Supplier<Procedure[]> branchingScheme;
4    public DFS(StateManager sm,Supplier<Procedure[]> b) {
5       this.sm = sm;branchingScheme = b;
6    }
7    public SearchStatistics solve() { return solve(stats -> false);}
8    public void solve(Predicate<SearchStatistics> limit) {
9       SearchStatistics stats = new SearchStatistics();
10      sm.withNewState( () -> {
11         try {
12            dfs(stats,limit);
13         } catch(StopSearch sx) {}
14      });
15   }
16   public void dfs(SearchStatistics stats,Predicate<SearchStatistics> limit) {
17      if (limit(stats))
18         throw new StopSearch();
19      Procedure[] branches = branchingScheme.call();
20      if (branches.length == 0)
21         notifySolution();
22      else
23         for (b : branches) {
24            sm.withNewState( () -> {
25               try {
26                  b.call();
27                  dfs();
28               }
29               catch(InconsistencyException e) {}
30            });
31         }
32   }
33 }
```

Listing 36: The Sequencing Combinator

```
1 public class Sequencer implements Supplier<Procedure[]> {
2     private Supplier<Procedure[]>[] branchingSchemes;
3     public Sequencer(Supplier<Procedure[]>... branchingSchemes) {
4        this.branchingSchemes = branchingSchemes;
5     }
6     @Override public Procedure[] get() {
7         for (int i = 0; i < branchingSchemes.length; i++) {
8             Procedure[] alts = branchingSchemes[i].get();
9             if (alts.length != 0)
10                return alts;
11        }
12        return Selector.EMPTY;
13    }
14 }
```

Listing 37: Using a Sequencer to Express a Two-Phased First-Fail Search.

```
1 IntVar[] a = ...;
2 IntVar[] b = ...;
3 ...
4 DFSearch dfs = makeDfs(cp,and(firstFail(a), firstFail(b)));
```

The rest of the implementation is depicted in Listing 38, which provides an excerpt of the class `BranchingScheme`. Lines 5–14 define a selector that receives, as inputs, an array `x`, a predicate `p`, and a function `f`: It selects an element `xi` of `x` such that `p(xi)` holds and `f(xi)` is minimal. Lines 16–29 define a search procedure that uses the first-fail principle to assign all variables in array `x`. It uses `selectMin` to find the free variable with the smallest domain and then branches in the same way as in the prior examples. Note Lines 30–32 that define a factory method `and` to create a sequencer combinator.

Draft September 15, 2018

Listing 38: Illustrating Selectors and Combinators

```
1  public class BranchingScheme {
2
3    [...]
4
5    public static <T,N> extends Comparable<N>>
6    T selectMin(T[] x,Predicate<T> p,Function<T,N> f) {
7       T sel = null;
8       for (T xi : x) {
9           if (p.test(xi)) {
10              sel = sel == null  f.apply(xi).compareTo(f.apply(sel)) < 0 ? xi : sel;
11          }
12      }
13      return sel;
14   }
15
16   public static Supplier<Procedure[]> firstFail(IntVar... x) {
17      return () ->  {
18         IntVar xs = selectMin(x,
19                               xi -> xi.size() > 1,
20                               xi -> xi.size());
21         if (xs == null)
22           return EMPTY;
23         else {
24           int v = xs.min();
25           return branch(() -> equal(xs,v),
26                         () -> notEqual(xs,v));
27         }
28     };
29   }
30   public static Supplier<Procedure[]> and(Supplier<Procedure[]>... choices) {
31      return new Sequencer(choices);
32   }
33   public static final Procedure[] EMPTY = new Procedure[0];
34   public static Procedure[] branch(Procedure... branches) { return branches;}
35 }
```

Draft September 15, 2018

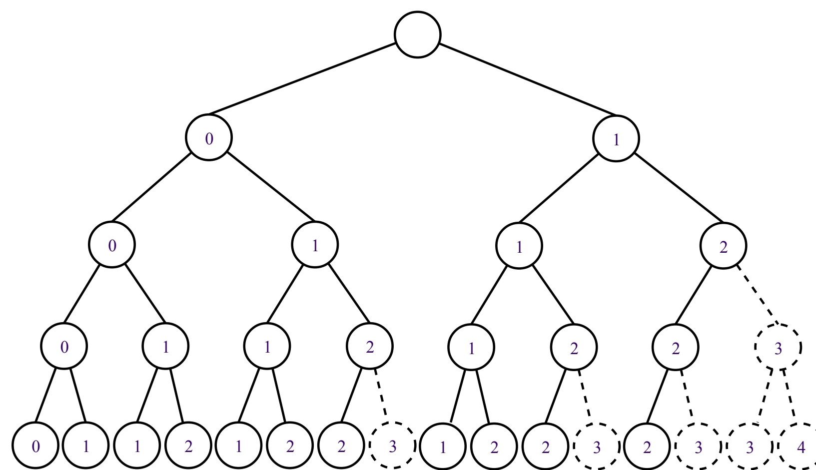

Fig. 6: Illustration of the nodes explored with a discprepancy limit of 2. The discrepancy is given in each node. Dashed nodes are pruned.

7.3 Limited Discrepancy Search

This section presents a combinator for Limited Discrepancy Search (LDS) [**?**]. LDS assumes that a reasonably good heuristic is available and explores by trusting the heuristic less and less over time. It often helps DFS from being stuck in unproductive parts of the search space. LDS explores the search in waves, allowing the search to differ from the heuristic by up to 0, 1, 2, . . . decisions in each wave. For instance, Figure 6 displays the leaves explored by LDS with a discrepancy limit of 2.

Listing 39: Limited Discrepancy Branching

```
1 public class LimitedDiscrepancyBranching implements Supplier<Procedure[]> {
2
3     private int curD;
4     private final int maxD;
5     private final Supplier<Procedure[]> bs;
6
7     public LimitedDiscrepancyBranching(Supplier<Procedure[]> bs, int maxDiscrepancy) {
8         this.bs = bs;
9         this.maxD = maxDiscrepancy;
10    }
11    @Override public Procedure[] get() {
12        Procedure[] branches = bs.get();
13        int k = Math.min(maxD - curD + 1, branches.length);
14        if (k == 0) return BranchingScheme.EMPTY;
15        Procedure [] branches_k = new Procedure[k];
16        for (int i = 0; i < k; i++) {
17            int bi = i;
18            int d = curD + bi; // branch index
19            branches_k[i] = () -> {
20                curD = d; // update discrepancy
21                branches[bi].call();
22            };
23        }
24        return branches_k;
25    }
26 }
```

Listing 40: Iterative Discrepancy Search

```
1 Objective obj = cp.minimize(totCost);
2 // Iterative Discrepancy Search
3 for (int dL = 0; dL < x.length; dL++) {
4    DFSearch dfs = makeDfs(cp,limitedDiscrepancy(firstFail(x),dL));
5    dfs.optimize(obj);
6 }
```

The Implementation of LDS in MiniCP is given in Listings 39 and 40. Listing 39 describes a combinator for implementing a wave, while Listing 40 presents the traditional iterative implementation of LDS. Any branching scheme can be wrapped into the LDS combinator to create a new branching scheme. The key is to create new closures from the embedded branching schemes. Each such closure updates the current discrepancy `curD` (line 20) before calling the actual branch. The current discrepancy is then used in line 14 to fail if the discrepancy limit `maxD` is exceeded. Note that the nodes explored in a given iteration of Listing 40 may overlap with the ones explored at the previous iteration. But this overlap is in general quite limited since most of them are pruned with the help of the primal bounds produced by the waves.

7.4 Large Neighborhood Search

Large Neighborhood Search (LNS) [23] is an effective search strategy to leverage the strengths of CP on large-scale applications. Its key idea is to improve an initial solution by fixing some of its decisions and searching for an improving solution when exploring the remaining search space. The sub-space exploration (often called a reconstruction phase) is often subject to a CPU time or failure limits to avoid being stuck. This process is repeated using randomization to select the subspaces. For instance, the subspace may be defined by fixing the assignments of a subset of the decision variables, the remaining variables being "relaxed", i.e., they can be assigned to any value in their domain. Table 2 highlights this process on the first four iterations of the quadratic assignment problem. It depicts the vector of decision variables and objective value. At each iteration, about 50% of the variables are randomly selected to be assigned to their value in the best-found solution, the other being "relaxed". CP then tries to improve the current best solution.

Draft September 15, 2018

iter 1	obj:28480											
best	0	1	2	3	4	6	9	5	8	10	11	7
relaxed	*	*	*	3	*	*	9	5	8	*	11	7
iter 2	obj:13456											
best	6	4	1	3	0	2	9	5	8	10	11	7
relaxed	6	*	1	3	*	*	*	5	*	*	*	7
iter 3	obj:13200											
best	6	4	1	3	9	2	0	5	11	10	8	7
relaxed	6	4	*	*	9	2	0	5	*	10	*	7
iter 4	obj:10792											
best	6	4	3	1	9	2	0	5	11	10	8	7
relaxed	6	*	*	*	9	*	*	*	11	10	*	*
...	...											

Table 2: The first four iterations of LNS on the quadratic assignment problem. At each LNS iteration, the current best solution and the randomized subspace. Relaxed variables are depicted with a * symbol.

Listing 41: Large Neighborhood Search

```
1  int[] xBest = IntStream.range(0,n).toArray();
2
3  dfs.onSolution(() -> {
4     // Update the current best solution
5     for (int i = 0; i < n; i++)
6         xBest[i] = x[i].min();
7     System.out.println("objective:"+objective.min());
8  });
9
10 for (int i = 0; i < nRestarts; i++) {
11    System.out.println("restart number #"+i);
12
13     dfs.optimizeSubjectTo(obj,stats -> stats.nFailures >= failureLimit, () -> {
14        // Assign the fragment 50% of the variables randomly chosen
15        for (int j = 0; j < n; j++) {
16           if (rand.nextInt(100) < 50) {
17              // after the optimizeSubjectTo those constraints are removed
18              Factory.equal(x[j], xBest[j]);
19           }
20        }
21      }
22    );
23 }
```

Listing 42: Large Neighborhood Search

```
1  public class DFS {
2    // remainder of the class (as before)
3    public void optimizeSubjectTo(Objective obj,Predicate<SearchStatistics> limit, Procedure
         subjectTo) {
4         sm.withNewState(() -> {
5             try {
6                 subjectTo.call();
7                 optimize(obj,limit);
8             } catch (InconsistencyException e) {}
9         });
10    }
11 }
```

An implementation of LNS in MiniCP is given in Listing 41. Line 1 defines an array $xBest$ to store the best-found solution. Lines 3–8 updates this array every time a new solution is found. Lines 10–23 performs `nRestarts` CP searches. Each such exploration starts by fixing about 50% of the variables to their values in the best-found solution (lines 15–20), before proceeding to the depth-first search itself. Method `optimizeSubjectTo` calls `withNewState` to preserve the initial state in which none of the

Draft September 15, 2018

variables are fixed. Therefore, at each iteration, this initial state is restored before starting fixing the selected variables. For completeness' sake, method `optimizeSubjectTo` is shown in Listing 42.

8 Teaching Material and Student Projects

MiniCP comes with a series of more than 20 implementation projects available on `https://bitbucket.org/minicp/minicp`. All the projects can be tested with junit-tests provided to students. These projects will teach students the material presented in this paper, as well as more advanced topics. They include

- Trailing and CP solver internal mechanisms by adding new useful functionalities to MiniCP (views, domain iterators, etc);
- Basic binary, logical, and reified constraints;
- Global constraints: table, element, allDifferent constraints;
- Problem specific and black-box heuristics, including LNS;
- Modeling good practices (redundant constraints, symmetry breaking, ...);
- Scheduling constraints;
- CP for frequent pattern mining, an emerging topic.

Slides that provide the required background to successfully implement the projects are also available. At the end of each slide deck, suitable implementation projects are identified. This material has been already used and tested at the ACP-2017 summer school and in the ING2365 course on CP at UCLouvain. The proposed projects are calibrated to take about 4 hours of implementation work for 12 weeks. In the future, some quizzes and homeworks will also be developed. A mini-solver competition was recently organized[3] for solving problems described by XCSP3 standardized modeling format. MiniCP comes with a parser and interface for the XCSP3 format to simplify participation. Students found the participation in a competition very motivating.

9 Conclusion

This paper introduced MiniCP, a light-weight, open-source constraint-programming solvers for educational purposes. Its goal is to stimulate the teaching and research in constraint programming by democratizing access to its core implementation concepts. MiniCP has a minimalist architecture with a close matching between theoretical and implementation concepts and a focus on compositionality and extensibility.

MiniCP is a living project already used in practice to teach constraint programming at some universities and in summer schools. It will continue to evolve and offer new teaching material. The project welcomes contributions and extensions under the form of pull requests on the source code, exercises, or as new reference to external pages. Feedback from Early adopters indicates that MiniCP is also useful for research purposes as it permits to experiment with new ideas easily in a clean and understandable, yet reasonably efficient, architecture.

References

1. Aggoun, A., Beldiceanu, N.: An Overview of the CHIP compiler. In: the 8th International Conference on Logic Programming (ICLP-91). The MIT Press, (Paris, France) (1991)
2. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, (revision a) (2012)
3. Boussemart, F., Lecoutre, C., Piette, C.: Xcsp3: An integrated format for benchmarking combinatorial constrained problems. arXiv preprint arXiv:1611.03398 (2016)
4. Colmerauer, A.: An Introduction to Prolog III. Commun. ACM **28**(4), 412–418 (1990)
5. Dynadec, Van Hentenryck, P., Michel, L., Schaus, P.: Comet v2. 1 user manual (2009)
6. Floyd, R.W.: Nondeterministic algorithms. Journal of the ACM (JACM) **14**(4), 636–644 (1967)
7. Hentenryck, P.V., Michel, L.: Constraint-based local search. The MIT press (2009)
8. van Hoeve, W.J., Katriel, I.: Global constraints. In: Foundations of Artificial Intelligence, vol. 2, pp. 169–208. Elsevier (2006)
9. Hooker, J.N.: Integrated methods for optimization, vol. 170. Springer Science & Business Media (2012)
10. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL-87. (Munich, Germany) (1987)
11. Knuth, D.E.: The art of computer programming: Volume 4B, Combinatorial Algorithms: Part 2, Backtrack Programming, vol. 4B. Adison-Wesley (2016)
12. Kuchcinski, K., Szymanek, R.: Jacop-java constraint programming solver. Procs. of CP Solvers: Modeling, Applications, Integration, and Standardization (2013)

Draft September 15, 2018

13. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Ibm ilog cp optimizer for scheduling. Constraints pp. 1–41 (2018)
14. Mackworth, A.: Consistency in Networks of Relations. Artificial Intelligence **8**(1), 99–118 (1977)
15. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. Constraints pp. 1–45 (2014)
16. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. Constraints **22**(2), 107–151 (2017). DOI 10.1007/s10601-016-9242-1. URL https://doi.org/10.1007/s10601-016-9242-1
17. van Omme, N., Perron, L., Furnon, V.: or-tools user's manual. Tech. rep., Technical report, Google (2014)
18. OscaR Team: OscaR: Scala in OR (2012). Available from https://bitbucket.org/oscarlib/oscar
19. Prud'homme, C., Fages, J.G., Lorca, X.: Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR **6241** (2014)
20. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search Combinators. Constraints **18**(2), 269–305 (2013). DOI 10.1007/s10601-012-9137-8. URL http://dx.doi.org/10.1007/s10601-012-9137-8
21. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. Handbook of Constraint Programming p. 493 (2006)
22. Schulte, C., Tack, G.: Views and iterators for generic constraint implementations. In: Recent Advances in Constraints (2005), *Lecture Notes in Artificial Intelligence*, vol. 3978, pp. 118–132. Springer-Verlag (2006). URL http://www.gecode.org/paper.html?id=SchulteTack:Advances:2006
23. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: International conference on principles and practice of constraint programming, pp. 417–431. Springer (1998)
24. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. The MIT Press, Cambridge, MA (1989)
25. Van Hentenryck, P., Carillon, J.P.: Generality Versus Specificity: An Experience with AI and OR Techniques. In: Proceedings of the American Association for Artificial Intelligence (AAAI-88). AAAI, Menlo Park, Calif., (St. Paul, MN) (1988)
26. Van Hentenryck, P., Deville, Y.: The Cardinality Operator: A New Logical Connective and its Application to Constraint Logic Programming. In: ICLP-91, pp. 745–759 (1991)
27. Van Hentenryck, P., Michel, L.: The objective-cp optimization system. In: International Conference on Principles and Practice of Constraint Programming, pp. 8–29. Springer (2013)
28. Van Hentenryck, P., Michel, L.: The Objective-CP Optimization System. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (2013)
29. Van Hentenryck, P., Michel, L.: Domain views for constraint programming. In: International Conference on Principles and Practice of Constraint Programming, pp. 705–720. Springer (2014)
30. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc (fd). The Journal of Logic Programming **37**(1), 139–164 (1998)

Draft September 15, 2018

	nodes	mini-cp	oscar-cp	choco
magic-series	1192	17	8	28
n-queens	49339391	537	295	328
QAP	144846	12	5	8^4
TSP	4285744	66	29	24
magic-square	3011034	41	23	25

Table 3: Performance comparison with Choco [19] and OscaR [18].

A Java Closures

Java 1.8 supports the concept of closures (aka first-order functions) which is at the core of functional programming. The MiniCP solver makes extensive use of closures for more readibility and conciseness. Consider the identity function $\lambda x.x$ expressed in lambda-calculus. It can be written in Java 1.8 as

Listing 43: An Identity Closure in Java

```
1 Function<Integer,Integer> r = (Integer x) ->  { return x;};
2 for(int i=0;i < 10;i++)
3     System.out.format("f(%d) = %d\n",i, r.apply(i));
```

The code indicates that Java provides a *generic type* `Function<A,B>` to represent a function of type $A \to B$. and r is used to refer to the identity function. The next two lines show how to `apply` closure r for various values. Since this example is somewhat verbose, Java 1.8 offers syntactic sugar to simplify the notation, dropping the argument type, the block syntax, and the `return` keyword to obtain

```
1 Function<Integer,Integer> r = x -> x;
```

This brings the definition of r close to a lambda calculus definition. Java 1.8 also makes it possible to define arbitrary closure types through the concept of functional interface. For instance, the snippet

Listing 44: A `Procedure` first-order function type

```
1 @FunctionalInterface public interface Procedure {
2     void call();
3 }
```

defines the type of a closure that takes no input and return no outputs. It just executes a block of code. MiniCP also makes extensive use of functional interfaces defined in the JDK 1.8. For instance, a *branching* (closure returning an array of `Procedure`) is none other than `Supplier<Procedure[]>`. Boolean predicates over some type T (i.e., first-order functions of type $T \to \mathbb{B}$ use `Predicate<T>` while first-order functions of type $T \to N$ use `Function<T,N>`.

B Performance Evaluation

The objective of those experiments is to measure the raw performances of the basic funtionalities offered by a solver: backtracking and propagation mechanisms, domains, search, etc. Our goal is to measure if, despite its simplicity, MiniCP achieve reasonable performance when compared to carefully engineered and optimized solvers, including Choco [19] implemented in Java and OscaR [18] implemented in Scala. Both [19] and [18] have more than 50K lines of code.

In state-of-the-art solvers, great care is dedicated to the efficient implementation of global constraints. Since comparing implementations of global constraints is not the purpose of this evaluation, the models used in the experiments and available here `https://www.dropbox.com/sh/5djpmt0rg5kr99p/AAAoghfC-1F04Qpz6xcgcy01a?dl=0`. are quite simple and composed of sum, elements, reification and binary constraints. The experimental evaluation ensures that the same search trees is explored by all the solvers. The evaluation also forces a sparse-representation of the domains as this is the only available option in MiniCP in the base implementation.

Table 3 presents the results. The performance of MiniCP is reasonably good despite its simplicity, flexibility and the fact that we explicitly refrain from optimizing the implementation. For instance, the code uses Java collections which induce a a significant overhead due to object and iterator creations; An optimized solver such as OscaR or Choco often uses its own array-based collections. Boolean variables in MiniCP are simply 0/1 integer variables, while OscaR uses a dedicated implementation. OscaR statically pre-allocates frequently used objects to avoid the dynamic creation of `StateEntries`. Aside from this lack of optimization in memory management, the performance difference can be explained by the fact that OscaR and Choco have a priority system to schedule light constraints before more complex ones. The default implementation of MiniCP has no constraint priorities.

Draft September 15, 2018