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

Mini-CP: A Minimalist Open-Source Solver to teach Constraint Programming

Laurent Michel · Pierre Schaus · Pascal Van Hentenryck

Received: date / Accepted: date

Abstract The success minisat solver has largely contributed to the dissemination of (CDCL) SAT solvers. Minisat has a neat and minimalist architecture that is well documented. We believe the CP community is currently missing such a solver that would permit new-comers to demystify the internals of CP technology. We introduce Mini-CP a white-box bottom-up teaching framework for CP implemented in Java. Mini-CP is voluntarily missing many features that you would find in a commercial or complete open-source solver. The implementation, although inspired by state-of-the-art solvers is not focused on efficiency but rather on readability to convey the concepts as clearly as possible.

1 Introduction

Many computer science students graduate without having heard about CP. One reason is that in most universities CP is simply not taught at all. The CP research community is mainly active in Europe (France) and Australia [1]. As a consequence, in Asia and America only a limited number people have ever heard about CP. In its latest volume when introducing backtrack programming, Knuth [5] does not even mention CP as a successful technique for solving combinatorial optimization problems. This appears to be symptomatic of the fact that CP does not disseminate well through the teaching at universities.

At CP2015 Frisch, McCreesh, Petrie, Prosser took the initiative to organize a workshop on teaching CP. The workshop ended with an open discussion to answer the question: *"As a community, what can we do to improve and increase the teaching of constraint programming?"*. The unanimous answer was that the community should communicate better and make teaching material more broadly available. Another important observation was that most CP teachers build their own teaching material without necessarily sharing it. The Association for Constraint Programming (ACP) has also identified the availability of teaching material as crucial for the future of CP as a research topic and tool adopted by practitioners. The ACP decided to promote the sharing of teaching material such that any university or professor who wants to propose a CP course can do it with a modest effort.

This work aims to fill this gap with the introduction of MiniCP. The hope is that with MiniCP any professor having a basic background in algorithmic can easily propose a CP course at his institution. MiniCP provides exercises, unit tests, and development projects. Ultimately, this educational solver will be accompanied by slides and video lectures that could be used in for flipped classroom.

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
Industrial Engineering Department, University of Michigan Ann-Harbor, MI USA
E-mail: pvanhent@umich.edu

[1] See `www.a4cp.org/cparchive/countries_by_year` for the CP publications countries by year

package	LOC
engine	867
reversible	301
examples	288
cp	154
search	148
tests	1316

Table 1: Packages of MiniCP and number of lines of codes (LOC) in each as computed by the command `sloccount`.

1.1 Influences

The design of MiniCP is influenced by several several solvers such as cc(fd) [17], Comet [3], Objective-CP [15], and OscaR [9]. Some inspiration is taken from the Microkernel introduced in [7]. Other solvers share similar design such as CP Optimizer [], OR-Tools [8], [6] and Choco [10]. MiniCP was implemented in Java 8 to make it accessible to the largest possible audience. The code makes extensive use of the functional interfaces introduced in Java 8 to model closures.

The closest related work is the chapter [12] from the handbook of CP. MiniCP is not a copy-based solver. Making an efficient copy-based solver requires some work when compared to a trail-based solver. It is an interesting exercise to consider the revisions needed to MiniCP's architecture to support a copy-based state management policy.

1.2 MiniCP in a Nutshell

Designing a CP solver is an exercise that demands to balance three opposite objectives, namely: Efficiency, Flexibility and Simplicity. This balancing exercise is often subjected to a bias as solver authors may wish to participate in competitions where performance is paramount or focus on application writing where flexibility is a key quality to adapt to idiosyncratic needs. While everyone agrees that simplicity is virtuous, it often ends up being sacrificed to the other objectives. The net result is a collection of solvers that can showcase wonderful performance and are used by experts to write sophisticated programs. Neophytes, occasional users and students are, unfortunately, collateral damage as solvers may prove difficult to effectively use. Studens are particularly vulnerable as the architectural decisions, key concepts and core abstractions often end-up being significantly obfuscated.

MiniCP is an *educational solver* that strikes a biased balance between Efficiency, Flexibility and Simplicity that clearly favors simplicity and elegance to neatly and compactly illustrate they key concepts that permeate every solver implementations. It is minimal as shown in Table 1 with a code base of 1500 lines of Java code (excluding modeling examples and unit-tests). Clearly, such a small footprint does not, by choice, include every feature found in mainstream solvers. Instead, it emphasizes clean, simple, easy to understand implementations allowing instructors and students to focus on core ideas and algorithms to get to the essence of a solver.

1.3 Target audience

MiniCP as a teaching framework that targets computer science students having a background in data-structures and algorithms. Students and instructors interested into teaching CP modeling language should consider the MOOC on Minizinc by Peter Stuckey [14] or a tutorial on XCSP3 format [1].

2 Educational Objectives

2.1 Topics

MiniCP is meant to serve as an exploratory vehicle to approach design and implementation decisions and shine a bright light on core aspects of a solver. It explores issues that pertain to the declarative models and the search. On the modeling front, it considers general inference mechanics that support constraint propagation, how to manage state, represent decision variables and reason with them and how to implement several classes of constraints including arithmetic, logical, reified and global constraints.

On the search front, it explores chronological backtracking with Depth-First-Search, variable and value selection heuristics, problem decomposition and large neighbhorhood search.

It is worth highlighting that MiniCP does *not* attempt to provide implementations for every possible issue explored by the Constraint Programming community. Some topics such as model reformulations and transformations, parallelization, alternate variables types –e.g., sets or continuous –, half-reified constraints, learning, or even symmetry-breaking to name just a few. It does not imply that these cannot be supported. Rather, these topics could be the objects of future extension or exploratory projects.

2.2 Learning outcomes

The learning outcomes fall in two categories. First, students completing modules based on MiniCP should grasp how declarative models and operational semantics of solvers meet and interact at *a macroscopic level*. Namely, they must understand their respective operational roles of inference and search and how they are blended by the solver. They should appreciate the role of search heuristics and their actual realization. Second, students should have acquired at a *microscopic level* a clear understanding of all the mechanics involved when solving a model. This encompasses low-level aspects related to state management and inference logic as well as low-level aspects related to the search. From a state and inference prospective, specific outcomes include:

- Trailing and state reversion
- Domain and variable implementation
- Propagation queue
- Arithmetic Constraints
- Logical Constraints
- Reified Constraints
- Global Constraints
- Views

While, from a search prospective, the outcomes include:

- Backtracking algorithms and depth first search
- Branch and Bound for Constraint Optimization
- Incremental Computation
- Variable and Value Heuristics implementation
- Searching with phases
- Large Neighborhood Search

2.3 Ancillary benefits

There are additional benefits for teaching CP in a Computer Science program that go beyond the sole motivation of learning/teaching CP.

Algorithms. By studying CP the student can really see and put in practice many of the algorithms and data-structures covered in a introductory algorithmic course (sorting, graph algorithms, dynamic programming, etc).

Architecture and design patterns The architecture of CP system is a good example of a layered architecture with many orthogonal concepts that are used together. CP is also a good example of re-usability. From a software engineering point of view, studying and extending a CP system is thus interesting. A CP system architecture is open for extension and re-usability. Many design patterns are used in CP systems like the observer, strategy, visitor, etc.

API Design CP systems are mainly intended for end-users. The API design strongly influence their perception and adoption. The way functionalities are exposed is very important for the end-user experience. Exposing students to API design issues is an effective way to convey the importance of elegant, intuitive and easy to use APIs.

Cross Benefits Finally, students who use and extend a CP system have the opportunity to learn many skills useful in every computational sub-disciplines. In particular, they get exposed to empirical methods (e.g., comparing algorithms or measuring performance), they must master testing and debugging skills in non-trivial software and finally they get yet another opportunity at wrangling with the analysis of algorithms and their space and time complexities.

3 Getting Started

MiniCP, as described in this paper, adopts the Constraint Programming mantra

$$CP = Model + Search$$

Namely, it espouses the view that a combinatorial optimization program using CP technology breaks down into a *model* and a *search* component. Programs written by end-users must embrace this point of view. This section illustrates a classic and simple application. It demonstrates how one can obtain an elegant and highly readable program for the well-known $n-$Queens problem.

Recall that $n-$Queens is the problem of placing n queens on an $n \times n$ checkboard so that no two queens can attack each other, i.e., no two queens can be on the same row, column, or diagonals. The model considered here uses a simple encoding. It relies on an array q of n *decision variables* (one per column) in which each decision variable q_i is meant to hold the row k in which the queen in column i is to be placed. An assignment to the decision variables is feasible if and only if it satisfies the constraints of the problem. By virtue of the encoding, one does not need to check that no two queens can be in the same column. However, one must enforce the following

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

Note how the first set of requirements simply states that two queens cannot be on the same row, the second states that two queens cannot be on the same downward diagonal and the third states that two queens cannot be on the same upward diagonal. The purpose of a *program* is, therefore, to find an assignment of values to the decision variables that satisfies all three sets of requirements. The program in Listing 1 contains two distinct parts: the declarative model in lines 1–9 and the search in lines 11–22. Several capabilities and design decisions are highlighted below.

Listing 1: N-Queens model in Mini-CP

```
1  int n = 8; // number of queens and size of board
2  Solver cp  = makeSolver();
3  IntVar[] q = makeIntVarArray(cp, n, n);
4  for (int i = 0; i < n; i++)
5     for (int j = i+1; j < n; j++) {
6        cp.post(notEqual(q[i], q[j]));
7        cp.post(notEqual(q[i], q[j], i-j));
8        cp.post(notEqual(q[i], q[j], j-i));
9     }
10
11 SearchStatistics stats = makeDFS(cp,
12  selectMin(q,
13    qi -> qi.getSize() > 1,
14    qi -> qi.getSize(),
15    qi -> {
16         int v = qi.getMin();
17         return branch(() -> equal(qi, v),
18                       () -> notEqual(qi, v));
19    }
20  )
21 ).onSolution(() -> System.out.println("solution:"+ Arrays.toString(q))
22 ).start();
```

The Declarative Model First, the code extensively uses the Factory design pattern to isolate and hide all the memory allocations. Line 2 instantiates a solver `cp` through a factory method `makeSolver`. At this stage, a solver simply encapsulates a model. Line 3 of the model instantiates a Java array q of n *decision variables* each with a domain of possible values $\{0..n-1\}$. The two subsequent nested loops consider all the pairs of indices $i, j \in 0..n-1$ with $i < j$ and state binary disequalities. Specifically, line 6 states that $q[i] \neq q[j]$, line 7 imposes $q[i] \neq q[j] + i - j$ to cover the down diagonal and line 8 imposes $q[i] \neq q[j] + j - i$ to cover the up diagonal. In all three cases, a factory method is also used to create a *constraint* and add it to the solver `cp` via a call to its `post` method.

The Search To understand this fragment, it is important to note that it heavily relies on *closures* to specify key search components. Consider that a search procedure is a search space splitting technique that inductively decomposes a search space $\mathcal{P}$ induced by a constraint set C into

$$\mathcal{P}(C) = \mathcal{P}(C \wedge q_i = v) \cup \mathcal{P}(C \wedge q_i \neq v)$$

since the two constraints $q_i = v$ and $q_i \neq v$ partition the search space. Exploring both sub-spaces inductively will ultimately deliver all solutions to the original problem. This is a partial specification as choosing both the index i of the variable to branch on, and the value v to use in the branching must also be provided. Choosing the variable is know as a *variable selection heuristic* whereas choosing a value v is a *value selection heuristic*. Naturally, if a variable is already bound to a single value (the size of its domain is 1), the variable is not eligible for branching.

Lines 11–22 define the *search procedure*. This is also predominantly declarative and addresses the specification of *branching eligibility*, *variable selection*, *value selection* and actual branching. The `selectMin` operator acts on the array q of decision variable. Its purpose is to produce a branching decision (here a pair of constraints $q_i = v$ and $q_i \neq v$). Following the classic first-fail principle, it must choose a queen q_i from array q that satisfies

$$q_i = \operatorname*{argmin}_{q_k \in q : |D(q_k)| > 1} |D(q_k)|$$

This *variable selection* is expressed in `selectMin` with two closures. Using lambda-calculus, the first closure simply captures the condition of the *argmin* and boils down to

$$\lambda x.|D(x)| > 1$$

The second closure encodes the optimality expression

$$\lambda x.|D(x)|$$

which the `selectMin` minimizes. The third and last closure (lines 15–18) expresses both the *value selection* and the *branching*. The fragment simply chooses the smallest value v in $D(x)$ to return an array of two constant closures which, upon execution, impose the constraints $x = v$ and $x \neq v$. Note how easy it would be to express alternative branching scheme such as $(x \leq m, x > m)$ where m is the median element in $D(x)$.

To wrap up the search specification, line 11 imposes the use of depth-first-search with the specified selector while line 21 provides a closure to execute for each solution found and line 22 actually kicks off the search with a call to the `start` method. Incidentally, the search returns an object `stats` that encapsulates a few statistics related to the search effort.

Discussion It is, perhaps, valuable to take a moment and note the key mechanisms at work. In MiniCP, the model is embedded in the solver. The construction of the model (or the search) hides all the memory management issues and relies on a factory design pattern instead. The search heavily relies on closures and Java 8 lambdas to specify the variable and value selection heuristics as well as the branching scheme and how to react to the production of a solution. This gives rise to a user-level program in which semantically related code fragment are lexically close to one another leaving the user with a sense of strong code cohesion. The overall program is succinct, elegant, very readable and does not betray any details about the actual implementation.

In essence, the user-level API fosters declarative reading for both the model creation and the search specification.

4 Semantics: A CP Solver

4.1 Purpose

The presentation of a full-featured solver can easily become overwhelming. This paper articulates a presentation through iterative refinements starting with simple concepts and progressively introducing new components and refining older ones as necessary. In particular, implementing a search process entails the management of a search space. This is doable in many ways and implementation details can too easily cloud key concepts. The section starts by ignoring state management issues and focuses on a presentation in which backtracking does not occur. A later refinement shows how state management can be completely modularized and isolated to the benefit of the system architecture. In particular, Section 4.2 describes the overall computational model, Section 4.3 discusses the fixpoint algorithm and Section 4.4 revisits the state management issues that arise in a backtracking search.

4.2 Computational model

A constraint satisfaction problem (CSP) is a triplet $\langle X, D, C\rangle$ where X is a set of decision variables, D is their associated domains and C is a set of relations defined over subsets of X. A constraint optimization problem (COP) is a quadruplet $\langle X, D, C, f\rangle$ in which f is, without loss of generality, a function defined over a subset of X to maximize or minimize. Clearly, the concepts of *variables*, *domains*, *constraints* and *objective* are essential to the specification of a CP model and the definition of the semantics of a solver. These concepts are introduced next.

Definition 1 (Domain) A domain D is, without loss of generality, a finite set of discrete values $D \subseteq \mathbb{Z}$.

A domain D supports several notions such as membership ($v \in D$), lower bound ($\min(D) = \min_{v\in D} v$), upper bound ($\max(D) = \max_{v\in D} v$) and relies on the total ordering borrowed from $\mathbb{Z}$.

Definition 2 (Decision Variable) A decision variable $x \in X$ has a domain D, denoted $D(x)$. It is instantiated (bound) when $|D(x)| = 1$, inconsistent when $D(x) = \emptyset$ and free when $|D(x)| \geq 2$.

Note how D is also used to denote the domain of a CSP. In that case D is really none other than the cross product of the domains of all the variables. Namely, $D = D(x_1) \times \cdots \times D(x_n)$ where the set of n variables of the CSP is $\{x_1, \cdots, x_n\}$.

Definition 3 (Constraint) A constraint $c \in C$ is a relation defined over a subset of k variables $\{x_1, \cdots, x_k\} = vars(c) \subseteq X$.

Definition 4 (Solution) Given a set of decision variables X, a solution σ is a domain D, such that $\forall x \in X : |\sigma(x)| = 1$

Definition 5 (Feasible Solution) Given a set of decision variables X, and a constraint set C, a feasible solution σ is a domain D, such that $(\forall x \in X : |\sigma(x)| = 1) \wedge \bigwedge_{c\in C} c(\sigma)$ holds.

Definition 6 (Solution Set) Given a CSP $\langle X, D, C\rangle$, the solution set $\mathcal{S}(\langle X, D, C\rangle)$ is the set of all feasible solutions to $\langle X, D, C\rangle$. Namely,

$$\mathcal{S}(\langle X, D, C\rangle) = \left\{ \sigma_i \in D : (\forall x \in X : |\sigma(x)| = 1) \wedge (\bigwedge_{c\in C} c(\sigma_i)) \right\}$$

Definition 7 (Bound Consistency) A $k-$ary constraint $c \in C$ is bound-consistent w.r.t. $D(x_1)$ through $D(x_k)$ if and only if, for every $i \in 1..k$, there exist values $v_j \in D(x_j)$ (with $j \neq i$) such that $c(v_1, \cdots, v_{i-1}, \min(D(x_i)), v_{i+1}, \cdots, v_k) \wedge c(v_1, \cdots, v_{i-1}, \max(D(x_i)), v_{i+1}, \cdots, v_k)$ holds.

Definition 8 (Domain Consistency) A $k-$ary constraint $c \in C$ is domain-consistent w.r.t. $D(x_1)$ through $D(x_k)$ if and only if, for every $i \in 1..k$, and for every value $v_i \in D(x_i)$, there exist values $v_j \in D(x_j)$ (for every $j \in 1..k : j \neq i$) such that $c(v_1, \cdots, v_{i-1}, v_i, v_{i+1}, \cdots, v_k)$ holds.

Without loss of generality, a constraint system C is said to be domain consistent w.r.t. D if and only if every constraint $c \in C$ is domain-consistent w.r.t. D[2]. Consistency notions are instrumental as they give rise to *filtering* algorithms associated to constraints. Filtering algorithms can be defined in a generic way. Practically though, they are often specialized to each constraint based on its semantics. Yet, they can be specified as follows.

Definition 9 (Consistent Filtering Algorithm) A filtering algorithm $\mathcal{F}$ for a constraint $c \in C$ is consistent if it does not remove feasible solutions $\mathcal{S}(\langle X, D, C\rangle) = \mathcal{S}(\langle X, \mathcal{F}_c(D), C\rangle)$

Clearly, a filtering algorithm for constraint c *contracts* the domains of variables appearing in c, getting rid of values that provably cannot appear in a solution and are thus inconsistent. Most of the filtering algorithms are monotonic in the sense that stronger input domains result in stronger output domains when executing the filtering algorithm for some constraint:

Definition 10 (Monotonic Filtering Algorithm) A filtering algorithm $\mathcal{F}$ for a constraint $c \in C$ is monotonic if $D_1 \subseteq D_2 \Rightarrow \mathcal{F}_c(D_1) \subseteq \mathcal{F}_c(D_2)$.

In some cases a neat mathematical characterization on on the resulting domains after executing a filtering algorithms is possible. In particular, a bound consistent one guarantees that the extreme values of the domains are part of a solution of the constraint:

Definition 11 (Bound Consistent Filtering Algorithm) A bound consistent filtering algorithm $\mathcal{F}$ for a constraint $c \in C$ is a consistent filtering defined over $vars(c) = \{x_1, \cdots, x_k\}$ with domains $D(x_1), \cdots, D(x_k)$ yields new domains $D'(x_1) \subseteq D(x_1), \cdots, D'(x_k) \subseteq D(x_k)$ for which c is bound consistent.

A domain consistent filtering guarantees that every-remaining values of the domains are part of a solution. This is the strongest filtering possible for a constraint:

Definition 12 (Domain Consistent Filtering Algorithm) A domain consistent filtering algorithm $\mathcal{F}$ for a constraint $c \in C$ is a consistent filtering defined over $vars(c) = \{x_1, \cdots, x_k\}$ with domains $D(x_1), \cdots, D(x_k)$ yields new domains $D'(x_1) \subseteq D(x_1), \cdots, D'(x_k) \subseteq D(x_k)$ for which c is domain consistent.

Efficiently implementing filtering algorithms is essential to a CP solver. To do so, one can define computational rules that govern the removal of values from domain. Consider the following

Example 1 (Bound Filtering specification $x = y + 1$.) To filter the constraint, one can simply consider the following inference rules

$$\begin{array}{rl} \max(D(y))^{\downarrow} & \Rightarrow \max(D(x)) \leq \max(D(y)) + 1 \\ \min(D(y))^{\uparrow} & \Rightarrow \min(D(x)) \geq \min(D(y)) + 1 \\ \max(D(x))^{\downarrow} & \Rightarrow \max(D(y)) \leq \max(D(x)) - 1 \\ \min(D(x))^{\uparrow} & \Rightarrow \min(D(y)) \geq \min(D(x)) - 1 \end{array} \tag{1}$$

stating, for instance, that whenever the upper bound of variable y decreases, one can compute in $\Theta(1)$ time a tighter upper bound for x's domain (the others are defined similarly).

It is equally straightforward to define filtering rules for a Domain Filtering Algorithm for the same constraint as demonstrated below

Example 2 (Domain Filtering specification $x = y + 1$.) The inference rules (1) plus the following fully specify the domain filtering algorithm for $x = y + 1$

$$\begin{array}{ll} v \notin D(y) & \Rightarrow v + 1 \notin D(x) \\ v \notin D(x) & \Rightarrow v - 1 \notin D(y) \\ |D(y)| = 1 & \Rightarrow D(x) = \{\min(D(y)) + 1\} \\ |D(x)| = 1 & \Rightarrow D(y) = \{\min(D(x)) - 1\}. \end{array}$$

[2] Bound consistency can be lifted similarly.

Events occurring on the domain of x or y appear in the antecedents of the implication rules. For instance, the event $v \notin D(x)$ refers to the disappearance of value v from $D(x)$. The occurrence of this event implies the loss of another value $v-1$ from $D(y)$ as dictated by the consequent of the rule. Note how the *event lexicon* granularity can have a significant impact on the specification of inference rules. Examples 1 and 2 rely on fairly fine-grained events. A coarser event such as `change(`$D(x)$`)` that is triggered whenever any changes occur in $D(x)$ would yield a different and coarse inference rule whose consequent would have to carry out a lot more work.

Choosing a suitable *event lexicon* is an impactful design decision that is be explored in Section 5 where the implementation of the filtering algorithms for constraints such as $x = y + c$ are discussed.

4.3 Fixpoint semantics

Given a constraint set C and a collection of filtering algorithms adapted to each constraint $c \in C$, one can define a domain D as the solution to the fixpoint equation

$$D = \bigcap_{c \in C} \mathcal{F}_c(D). \tag{2}$$

From a computational standpoint, the fixpoint is reached through an iterative process that yields a sequence of contracting domains ultimately yielding a solution to the fixpoint equation. This iterative process is sketched in Algorithm 1.

An important property of the greatest fixpoint computed is that given the filtering for the constraints are monotonic, the fixpoint is the largest one and it is unique [13]. The order in which the constraints are considered thus doesn't matter.

Algorithm 1: Fixpoint algorithm

Data: D, C
Result: D the solution to the fixpoint equation (2)

```
1 fix ← false;
2 while ¬fix do
3 |   fix ← true;
4 |   foreach c ∈ C do
5 |   |   D' ← F_c(D);
6 |   |   if D' ≠ D then
7 |   |   |   D ← D';
8 |   |   |   fix ← false;
```

Definition 13 (`failure` and `success` predicates) Let $\mathcal{F}_C$ denote the fixpoint computation function over a constraint set C. Given an initial domain D_0, this function produces a domain $D_1 = \mathcal{F}_C(D_0)$ that satisfies the fixpoint equation (2).

- If $\exists x \in X : |D_1(x)| = 0$, D_1 is inconsistent and the constraint set is infeasible w.r.t. D_0. In which case, the predicate `failure`(D_1) is true.
- If $\forall x \in X : |D_1(x)| = 1$, D_1 is consistent and denotes a feasible assignment. In which case, the predicate `success`(D_1) is true.

4.4 Decomposition

Inference on a domain D_0 yields $D_1 = \mathcal{F}_C(D_0)$ and concludes with one of three outcomes:

1. `failure`(D_1) implies an infeasible constraint set in the space denoted by D_1 and the initial space D_0 can be discarded.
2. `success`(D_1) implies a feasible solution which can be set aside and reported.
3. $\neg$`failure`$(D_1) \wedge \neg$`success`(D_1) implies a potentially feasible space D_1 which may contain solutions and must be explored.

In the third case, it is necessary to decompose the search space to isolate the solutions. To this end, one can use a *branching scheme* following the well known divide and conquer strategy.

Definition 14 (Branching Scheme) Given a domain D over a set of variables X, a branching scheme is a list of constraints $(c_1, \cdots, c_k)$ such that

$$\mathcal{S}(\mathcal{F}_{C \wedge c_1}(D)) \cup \cdots \cup \mathcal{S}(\mathcal{F}_{C \wedge c_k}(D)) = \mathcal{S}(\mathcal{F}_C(D)) \quad (3)$$

$$\mathcal{S}(\mathcal{F}_{C \wedge c_i}(D)) \cap \mathcal{S}(\mathcal{F}_{C \wedge c_j}(D)) = \emptyset \quad \forall i \neq j \in 1..k \quad (4)$$

Namely, it creates a clean partition of the solution set (equation 4) that covers the entire search space (equation 3).

Example 3 (Minimum domain branching) Given a variable $x \in X$ with $D(x) = \{1..10\}$, the branching scheme $(x = 1, x \neq 1)$ is a covering binary partition of the search space. It corresponds to the classic minimum domain value branching.

Example 4 (Dichotomic branching) Given a variable $x \in X$ with $D(x) = \{1..10\}$, the branching scheme $(x \leq 5, x > 5)$ is a covering binary partition of the search space.

With a branching scheme in hand, the exploration of the search space for a CSP $\langle X, D, C \rangle$ can be expressed with the pseudo-code shown in Algorithm 2. The `solve` function takes a CSP and produces a set S of feasible solutions. It operates on a queue Q of CSPs and the structural invariant is

$$\mathcal{S}(D) = S \cup \bigcup_{\langle X_i, D_i, C_i \rangle \in Q} \mathcal{S}(D_i)$$

Namely, no solutions are ever lost and solutions move from their implicit representation through the CSPs in the queue Q to the explicit list of feasible solutions in S. Observe how each iteration of the loop extracts a CSP $\langle X_0, D_0, C_0 \rangle$, obtains a branching scheme $(c_1, \cdots, c_k)$ with k constraints. Line 8 simply iterates on the branching constraints and computes the fixpoint $\mathcal{F}$ on $C \wedge c_i$ to filter the search space D_0 to $D_i \subseteq D_0$. If the outcome is a success, the solution is added to S. If it is a failure, the space D_i is discarded. Otherwise, a new sub-CSP is added to Q to be further explored.

Algorithm 2: Generic Search in MiniCP

```
   Data: X, D, C
   Result: S⟨X, D, C⟩
1  S ← ∅ ;
2  if success(D) then
3      return {D} ;
4  Q ← {⟨X, D, C⟩} ;
5  while Q ≠ ∅ do
6      ⟨X0, D0, C0⟩ ← deQueue(Q) ;
7      (c1, ··· , ck) ← branching(X0, D0);
8      foreach i ∈ 1..k do
9          Di ← F_{C∧ci}(D0) ;
10         if success(Di) then
11             S ← S ∪ {Di}
12         else if failure(Di) then
13             continue;
14         else
15             enQueue(Q, ⟨X0, Di, C0 ∧ ci⟩) ;
16 return S ;
```

An actual implementation should consider how to efficiently represent all the necessary data structures and avoid costly state replication. This is explored in Section 6.

5 Inference Implementation

This section describes the implementation of domains in section 5.1, variables in section 5.2, constraints in section 5.3 and the fixpoint computation in section 5.4.

For the sake of conciseness we have omitted all the methods related to upper and lower bounds modifications (on the domains, variables, etc) to only focus on assignment and value removals[3].

[3] We invite the interested reader to consult the source code of MiniCP for an implementation of these methods

5.1 Domain

A `SparseSet` is data structure for set representation particularly convenient for domains implementation [11]. It is initialized as full set of n values from 0 to $n-1$ and only permits the removal of values (no insertion possible thus). Given a set with k values, the main advantages of this data structure are

1. a constant time complexity for value removal,
2. a constant time for the removal of all the values except one (assignation operation), and
3. an iteration in $O(k)$ over the k values in the set

The state of a sparse set is defined by the instance variables given in Listing 2.

Listing 2: SparsetSet Structure

```
1 public class SparseSet {
2   private int [] values;
3   private int [] indexes;
4   private int size;
5   private int n;
6   public boolean remove(int val) {...}
7   public void removeAllBut(int v) {...}
8   public boolean contains(int val) { return indexes[val] < size;}
9 }
```

The class invariant for a `SparseSet` is:

- An array `values` of size n. It contains a permutation of the numbers $[0..n-1]$ such that the *size* first entries (prefix) are the values *in* the set, and the $n - size$ last ones are *removed* from the set.
- An array `indexes` of size n. It gives the position for each value in the *value* array: `values`$[$`indexes`$[v]] = v, \forall v \in \{0..n-1\}$.

An initial sparse-set of 9 values is depicted on Figure 1.

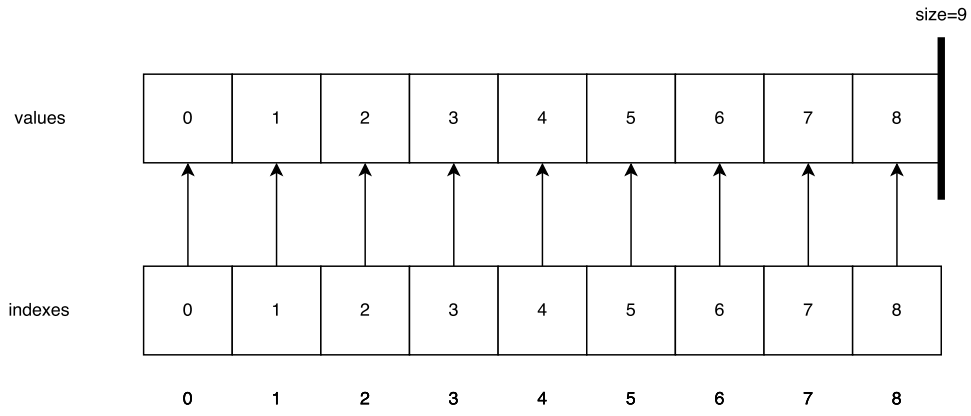

Fig. 1: Sparset Set

Maintaining the class invariant for the value removal operation is simple: exchange the value that needs to be removed with the one of the last position before decrementing the size. The two corresponding entries in `indexes` are updated to reflect this exchange of positions. Figure 6.2.1 illustrates the internal state after the removal of the value `4` followed by the removal of value `6`.

The removal of every value except the value v is done in $O(1)$ by placing the value v in first position with one swap operation between the value in first (`values[0]`) and v and updating `indexes` to reflect the new positions before updating `size=1`.

The domain API is given in Listing 3. It allows to pass a domain listener to the update operations as proposed in [16]. This listener is notified in case the domain has lost some value(s) or if it becomes a singleton. As will be seen in Section 5.2, this listener mechanism is instrumental for implementing the filtering rules triggered based on antecedents as introduced in Section 4.2.

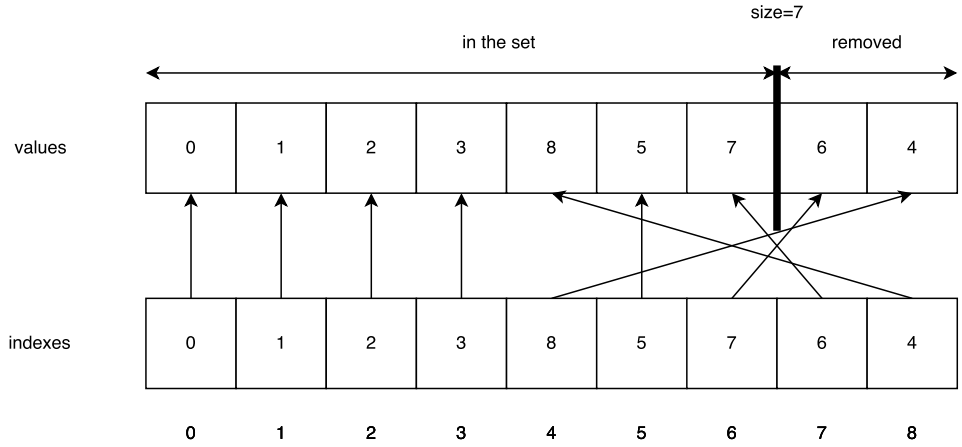

Fig. 2: Sparset Set

Listing 3: Domain API

```
1  public interface DomainListener {
2      void bind();
3      void change(int domainSize);
4  }
5  public abstract class IntDomain {
6      public abstract int getMin();
7      public abstract  int getMax();
8      public abstract  int getSize();
9      public abstract  boolean contains(int v);
10     public abstract  boolean isBound();
11     public abstract  void remove(int v, DomainListener x) throws InconsistencyException;
12     public abstract  void removeAllBut(int v, DomainListener x) throws
           InconsistencyException;
13 }
```

An implementation fragment of a domain using an internal `SparseSet` is given in Listing 4. The implementation mostly delegates the update operations to the internal sparse-set as can be seen in the body of the `remove` method. One can observe that the listener methods are correctly notified in case the value is removed (`x.change()`) and in case only one value remains (`x.bind()`).

Listing 4: Domain Implementation

```
1  public class SparseSetDomain extends IntDomain {
2      private SparseSet domain;
3      private int offset;
4      public SparseSetDomain(Trail trail, int min, int max) {
5          offset = min;
6          domain = new SparseSet(trail, max-min+1);
7      }
8      public int getMin() { return domain.getMin() + offset; }
9      public int getMax() { return domain.getMax() + offset; }
10     public int getSize() { return domain.getSize(); }
11     public boolean contains(int v) { return domain.contains(v - offset); }
12     public boolean isBound() { return domain.getSize() == 1; }
13     public void remove(int v, DomainListener x) throws InconsistencyException {
14         if (domain.contains(v - offset)) {
15             boolean maxChanged = getMax() == v;
16             boolean minChanged = getMin() == v;
17             domain.remove(v - offset);
18             if (domain.getSize() == 0) throw new InconsistencyException();
19             x.change(domain.getSize());
20             if (domain.getSize() == 1) x.bind();
21         }
22     }
23     ...
24 }
```

5.2 Variables

The `IntVar` API is given in Listing 5. Every concrete variable implementation implements this interface.

Listing 5: IntVar

```
1  public interface IntVar {
2      Solver getSolver();
3      void propagateOnDomainChange(Constraint c);
4      void propagateOnBind(Constraint c);
5      void whenDomainChange(ConstraintClosure.Closure c);
6      void whenBind(ConstraintClosure.Closure c);
7      int getMin();
8      int getMax();
9      int getSize();
10     boolean isBound();
11     boolean contains(int v);
12     void remove(int v) throws InconsistencyException;
13     void assign(int v) throws InconsistencyException;
14 }
```

A `Constraint` is an object with a single method `propagate` that filters domains. The `Constraint` objects will be detailed in section 5.3. A variable can attach constraints to be notified when its domain changes with the methods `propagateOnDomainChange` and `propagateOnBind`. These methods correspond to the definition of antecedent filtering rules introduced in Section 4.2:

$$\begin{array}{ll} \texttt{change}\,(D(x)) & \Rightarrow c.propagate() \\ |D(x)| = 1 & \Rightarrow c.propagate() \end{array}$$

One can also provide to `whenDomainChange` and `whenBind` a closure. This closure is internally wrapped into a constraint that delegates its `propagate` implementation to a call to the closure.

A possible implementation of `IntVar` interface is given in Listing 6. As expected `IntVarImpl` encapsulates an `IntDomain` object. The stacks `onDomainChange` and `onBindChange` contain all the constraints interested to be awakened whenever a domain modification or a biding event occurs. Whenever such an event occurs, the scheduling of the constraints is done using though the `DomainListener` mechanism.

Listing 6: IntVar Implementation

```
1  public class IntVarImpl implements IntVar {
2
3      Solver cp;
4      IntDomain domain;
5      Stack<Constraint> onDomain;
6      Stack<Constraint> onBind;
7
8      private DomainListener domListener = new DomainListener() {
9          public void bind() { scheduleAll(onBind); }
10         public void change(int domainSize) { scheduleAll(onDomain); }
11     };
12     public IntVarImpl(Solver cp, int min, int max) {
13         if (min > max)
14             throw new InvalidParameterException("empty domain");
15         this.cp = cp;
16         cp.registerVar(this);
17         domain = new Domain(cp.getTrail(),min,max);
18         onDomain = new Stack<>(cp.getTrail());
19         onBind  = new Stack<>(cp.getTrail());
20     }
21     private void scheduleAll(ReversibleStack<Constraint> constraints) {
22         for (int i = 0; i < constraints.size(); i++)
23             cp.schedule(constraints.get(i));
24     }
25     public void whenDomainChange(ConstraintClosure.Filtering c) {
26         onDomain.push(new ConstraintClosure(cp,c));
27     }
28     public void whenBind(ConstraintClosure.Filtering c) {
29         onBind.push(new ConstraintClosure(cp,c));
30     }
31     public void propagateOnDomainChange(Constraint c) {
32         onDomain.push(c);
```

```
33      }
34      public void propagateOnBind(Constraint c) {
35          onBind.push(c);
36      }
37      public void remove(int v) throws InconsistencyException {
38          domain.remove(v, domListener);
39      }
40      public void assign(int v) throws InconsistencyException {
41          domain.removeAllBut(v, domListener);
42      }
43  }
```

Let us summarize the sequence of events triggered whenever the domain of a variable is modified through the `remove` method:

1. The value is removed from the domain contained in the `IntVarImpl` by passing the `domainListener` object.
2. The `change` method of the `domainListener` implemented in `IntVarImpl` is called and consequently all the constraints present in `onDomain` stack are scheduled in the propagation queue of the `Solver`.
3. If after this removal the domain becomes a singleton then the `bind` method of the `domainListener` awakens also the constraints contained in `onBind` stack.
4. An `InconsitencyException` is thrown from the domain in case of removal of the last value in the domain. This exception is voluntarily not caught to let the search fail and backtrack as discussed later in Section 6.

5.3 Constraints

Every constraint extends the Constraint class and must implement the `post` and method plus possibly override the `propagate` one:

1. The `post` method checks that it is consistent according to its definition and the current domains. It can also do a first propagation to remove inconsistent values. It is also in this method that the constraint registers for domain modifications of the variables in its scope. When a constraint is posted in a model (like for instance with the `cp.post(noEqual)` in Listing 1), the post method of the constraint is called. If the constraint does not take the chance to register to domain modification at this time, the propagate method of the constraint will never be called. The post method is thus in charge of putting in place the glue between the variables and the constraint.
2. The `propagate` method removes inconsistent values. Assuming the constraint registered (in its post method) to domain modification events of the variables in its scope, It is called when a domain of a variable has been modified and so the constraint can potentially remove new inconsistent values or detect an inconsistency. The propage method is really the entry point of the filtering algorithm that implements the constraint. Many implementations of the `post` method terminate with a call to the `propagate` method.

If any inconsistency is detected (the constraint cannot be satisfied), then a `Inconsistency` exception must be thrown.

We exemplify the constraint implementation with the constraints used in the N-Queens model of Listing 1. The $x \neq y + c$ constraint states that x must take a value different from $y + c$. The inference rules are for this constraint are:

$$\begin{array}{l} |D(y)| = 1 \Rightarrow \min(D(y)) + c \notin D(x) \\ |D(x)| = 1 \Rightarrow \min(D(x)) - c \notin D(y) \end{array}.$$

The code for this constraint implementing these inference rules is given in Listing 7.

Listing 7: NotEqual Constraint

```
1  public class NotEqual extends Constraint {
2
3      IntVar x, y;
4      int c
5
6      public NotEqual(IntVar x, IntVar y, int c) { // x != y + c
7          super(x.getSolver());
```

```
        this.x = x;
        this.y = y;
        this.c = c;
    }
    @Override
    public void post() throws InconsistencyException {
        if (y.isBound())
            x.remove(y.getMin() + c);
        else if (x.isBound())
            y.remove(x.getMin() - c);
        else {
            x.whenBind(() -> y.remove(x.getMin() - c));
            y.whenBind(() -> x.remove(y.getMin() + c));
        }
    }
}
```

5.4 Fixpoint computation

The `Solver` class given in Listing 8 deals with the propagation and the fix-point computation. It contains a stack with all the constraints that have been scheduled and need to be propagated.

The `fixPoint` method pops the next constraint from the propagation queue, calls its `propagate` method. Keep in mind propagating a constraint by cause other constraints to be scheduled in the propagation queue. For efficiency reason, it would be useless to schedule a same constraint twice in the propagation queue. Therefore each constraint contains a boolean flag denoted `scheduled` which is set to true whenever the constraint is added in the queue and unset when the constraint is removed from the queue. The fixpoint computation finishes when the queue becomes empty.

The model declaration in MiniCP done by posting (i.e. adding) constraints using the `post` method of the `Solver`. Posting a constraint has two effect:

- It calls the post method on the constraint which checks its consistency and registers to the antecedents of the filtering rules.
- It computes the fixpoint computation on the set of already posted constraint and the newly added constraint.

Listing 8: Solver class

```

public class Solver {

    private Stack<Constraint> propagationQueue = new Stack<>();

    public void schedule(Constraint c) {
        if (!c.scheduled && c.isActive()) {
            c.scheduled = true;
            propagationQueue.add(c);
        }
    }

    public void fixPoint() throws InconsistencyException {
        boolean failed = false;
        while (!propagationQueue.isEmpty()) {
            Constraint c = propagationQueue.pop();
            c.scheduled = false;
            if (!failed) {
                try { c.propagate(); }
                catch (InconsistencyException e) {
                    failed = true;
                }
            }
        }
        if (failed) throw new InconsistencyException();
    }

    public void post(Constraint c) throws InconsistencyException {
        c.post();
        fixPoint();
```

```
31     }
32 }
```

6 Search Implementation

6.1 Depth First Search

The generic search described in Algorithm 2 does not dictate any search strategy. Indeed, different ordering policies for the queue Q lead to different search strategies. Depth First Search is conceptually simple, practically useful and ubiquitous in state-of-the-art solvers. It arises from Algorithm 2 by simply relying on ordering CSPs in Q by their depth and insertion order. Consider the search tree depicted in Figure 3. It corresponds to a search space with 3 binary variables and no constraints where each leaf (in green) is a potential solution. DFS starts with the leftmost path in the tree, producing the solution $[0, 0, 0]$ then backtracks to its most recent decision that it revises to assign the value 1 to the variable producing the solution $[0, 0, 1]$. The choice of which variable and value to branch on can change the topology of the tree. When constraints are present these choices can have a major impact on the *size of the tree* and therefore how quickly one can find a solution.

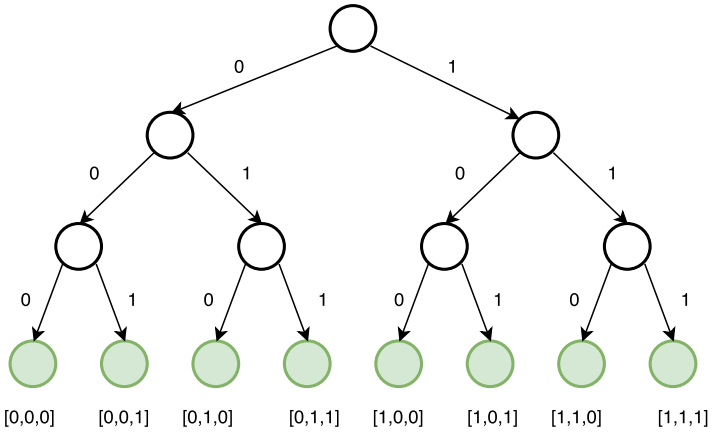

Fig. 3: A Simple Complete Binary Search Tree

In its purest form, DFS can be expressed as a recursive method as illustrated in Listing 9. The recursive method `dfs` relies on generic capabilities such as `branching` responsible for delivering the branching decisions. It can be easily externalized by adopting the Java 8 lambda architecture. Indeed, `branching` is a lambda that implements the *functional interface* called `Choice` and shown at the top of Listing 9. Note how the class constructor receives and holds in an instance variable a reference to the externally defined brancher. Interestingly, each alternative produced by branching is yet another lambda implementing the `Alternative` functional interface also shown in Figure 9. To embrace the selected alternative a, it suffices to call its lambda. Note that this lambda takes no input and returns no results as it will achieve its objective through side-effects. When there is nothing left to branch on, `branching` returns an empty list (`alternatives.length == 0`) and the generic method `notifySolution` is invoked to notify the caller that a solution is found.

Listing 9: Core DFS Skeleton

```
1  @FunctionalInterface
2  public interface Choice {
3      Alternative[] call();
4  }
5  @FunctionalInterface
6  public interface Alternative {
7      void call();
8  }
9  public class DFS {
10     private Choice branching;
11     public DFS(Choice b)  { branching = b;}
```

```
12    public void dfs() {
13       Alternative[] alternatives = branching.call();
14       if (alternatives.length == 0)
15          notifySolution();
16       else
17          for (a : alternatives) {
18             a.call();
19             dfs();
20          }
21    }
22 }
```

It is perhaps useful to realize that this skeleton completely isolates the search logic from what the alternatives, i.e., the branches, actually do. In particular, there is no provision in this template to manage the state of the solver manipulated by the search. For instance, when the invocation of an alternative entails posting a constraint, returning from a recursive call to `dfs` should *retract* the constraint that was added just prior. CP solver have adopted different tactics to support state management functions. Some solvers, e.g., Gecode [**?**] use copying to preserve the state and restore it. Others, e.g., cc(fd) [17], CPO [**?**], Comet [3], Jacop [6], Choco [10], Sicstus-Prolog [2], Or-Tools [8], OscaR [9], and Objective-CP [15] rely on *trailing* instead which is discussed next.

6.2 State Management

Trailing is a relatively simple state management tactic that facilitates the restoration of the computation state to an earlier version, effectively *undoing* changes that were imposed since then. Knuth [5] attributes to Floyd [4] the first formulation of this mechanism and its usage in a backtracking algorithm.

MiniCP provides two abstractions to adopt a restorable state, namely, the `ReversibleInt` and `ReversibleBool` classes. From an end-user standpoint, both classes are simple abstract data types that encapsulate, respectively, an integer and a boolean. The interfaces supported by both classes allow user to retrieve and set the encapsulated value. In sense, both simply represent mutable objects[4]. Perhaps far more importantly, both classes provide a key state management capability. Specifically, any change done to the state is *logged* against a state manager object responsible for reverting the changes should the user wish to restore an earlier computation state. Consider the reversible integer abstraction in Listing 10. It exposes four simple methods to get and set the value of the encapsulated value. From an end-user prospective, they behave exactly like vanilla integers.

Listing 10: Reversible Integer Interface

```
1 public interface RevInt {
2     int setValue(int v);
3     int getValue();
4     int increment();
5     int decrement();
6 }
```

An implementation for this interface must provide the services described above, it must also deliver logic needed to provide restorability. It is certainly clear that changes to the state of a solver occur in batches and one typically does not need to cherry pick which change to restore but instead adopts wholesale state restoration where an entire batch of changes is *undone*. This is achievable with an object responsible to track the changes that occur over time and group them in meaningful batches. Consider a search procedure proceeding down a path in a search tree (e.g., the tree in Figure 3). At each node of the tree, the solver might make a number of changes to the state to head down the left branch. When returning to the parent node in order to go down the right branch, one should first undo those changes and restore the state to the pristine form it had when entering the parent node the first time. This property is true at every node of tree. In particular, the batches of changes along a path from the root to a leaf are *stacked* so that returning to a shallower node along the path entails undoing the changes in all the batches stacked on top of the node we wish to reach.

Unsurprisingly, the `Trail` abstract data type provides a minimalistic API mean to record changes on the trail and manage stacked batches of changes. Its implementation is equally straightforward and is captured in terms of two simple stacks as depicted on Figure 4. The first stack `trail` holds entries capable to undo any change. Entries are clearly polymorphic to adapt to various reversible entities (e.g.,

[4] Mutability sets them apart from the builtin Java classes `Integer` and `Boolean`.

integers, booleans, etc). The second stack `trailLimit` is meant to track the batches. It is a simple stack of integers where each integer represent the height that the `trail` stack should have once all changes are undone. Namely, to restore the most recent batch, it is sufficient to apply the entries between the top of the `trail` stack and the top value recorded at the top of the `trailLimit` stack.

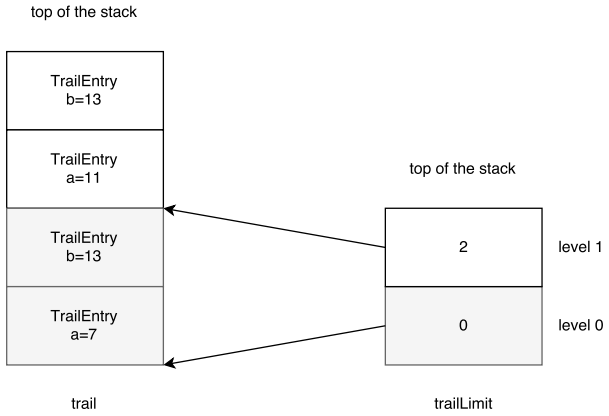

Fig. 4: For each trail entry, we give the identifier it refers to and the value that would be restored.

Consider Listing 13 in which one declares two reversible integers a and b with initial values 7 and 13. A call to `trail.push` creates an initial batch. It is followed by two writes to a and b. The second call to `trail.push` creates a second batch and is again followed by two writes to a (4) and b (9). Figure 4 depicts the state of the trail at this stage of the execution. A call to `trail.pop` would undo the top-most batch and restore a to 11 and b to 14. A second pop would restore the initial state.

Listing 11: Trail and ReversibleInt Manipulation

```
1  Trail trail = new Trail();
2  ReversibleInt a = new ReversibleInt(trail,7);
3  ReversibleInt b = new ReversibleInt(trail,13);
4
5  trail.push();      // Conceptually: record current state a=7, b=13
6     a.setValue(11);
7     b.setValue(14);
8     trail.push();   // Conceptually: record current state a=11, b=14
9        a.setValue(4);
10       b.setValue(9);
11    trail.pop();
12 trail.pop();
```

It is worth noting that *conceptually*, the push operation records the current state. *Operationally* though, it merely creates a batch that will be the receptacle for changes to come. In practice, it is the pop operation that restores *only the parts that have changed*. This is made possible by upgrading the implementation of the setter methods (on a and b) to record on the trail the value about to be obliterated by the setter with a call to `trail.pushOnTrail`. The result is a lazy recording strategy where one only pays a cost linear in the number of changes being exacted on the state. In particular, it is possible to use a time stamping technique to avoid recording multiple changes to the same state variables within the same batch. Indeed, this would be wasteful as only the first write needs to be restored. The reversible integer implementation is a class that bundles the behaviors dictated by the `RevInt` interface with the trail-based restoration just discussed. Its implementation is shown unabridged in Listing 12.

Listing 12: TrailEntry Interface

```
1  public interface TrailEntry {
2      public void restore();
3  }
4
5  public class ReversibleInt implements RevInt {
6      class TrailEntryInt implements TrailEntry {
7          private final int v;
8          public TrailEntryInt(int v) { this.v = v;}
```

```
9           public void restore()       { ReversibleInt.this.v = v;}
10      }
11      private Trail trail;
12      private int v;
13      private Long lastMagic = -1L;
14
15      public ReversibleInt(Trail trail, int initial) {
16          this.trail = trail;
17          v = initial;
18          lastMagic = trail.magic;
19      }
20      private void trail() {
21          long trailMagic = trail.magic;
22          if (lastMagic != trailMagic) {
23              lastMagic = trailMagic;
24              trail.pushOnTrail(new TrailEntryInt(v));
25          }
26      }
27      public int setValue(int v) {
28          if (v != this.v) {
29              trail();
30              this.v = v;
31          }
32          return this.v;
33      }
34      public int getValue()  { return this.v; }
35      public int increment() { return setValue(getValue()+1);}
36      public int decrement() { return setValue(getValue()-1);}
37  }
```

Note how the `ReversibleInt` constructor takes as input the state management class *trail* as well as an initial value. Its `getValue` method is straightforward and its convenience methods (`increment` and `decrement`) are equally simple. The `setValue` method devolves into a 'noop' when one does not actually change the value. If, however, v differs from the current state, the object *trails* its current state and then overwrites it with a new value. Trailing consists of adding an entry on the trail stack with a call to `trail.pushOnTrail`. The entry being pushed is polymorphic (to uniformly handle all types of reversible) and the concrete entry is shown at the top of the listing. It is a simple instance of a nested class providing a `restore` method meant to revert the state to the saved value. Lastly, the implementation adopts the idea of *semantic trailing* by simply augmenting the state with a timestamp ($lastMagic$) that monotonically increases with each call to `push` on the trail and is tested to avoid adding unnecessary entries if a reversible has already been trailed in this batch. To complete the discussion, it suffices to illustrate the trail implementation with Listing 13.

Listing 13: Trail

```
1  public class Trail {
2      public long magic = 0;
3      private Stack<TrailEntry> trail = new Stack<TrailEntry>();
4      private Stack<Integer>    trailLimit = new Stack<Integer>();
5      public void pushOnTrail(TrailEntry entry) { trail.push(entry);}
6      public void push(){
7          magic++;
8          trailLimit.push(trail.size());
9      }
10     public void pop() {
11         int n = trail.size() - trailLimit.pop();
12         for (int i = 0; i < n; i++) trail.pop().restore();
13         magic++;
14     }
15 }
```

6.2.1 Domain Reversibility

The domain of a variable needs to be restored on backtrack when a `pop` operation occurs on the trail. This reversibility is obtained by using a `ReversibleSparsetSet` data structure instead of a `SparsetSet`. The only difference between a `ReversibleSparsetSet` and `SparsetSet` is the use of `ReversibleInt` in to store the `size`.

Listing 14: ReversibleSparsetSet Structure

```
1 public class ReversibleSparseSet {
2   private int [] values;
3   private int [] indexes;
4   private ReversibleInt size;
5   private int n;
6 }
```

As a consequence after the `pop` operation, `size` recovers its previous value and the removed values are back in the set. For instance, assume a push was performed on the trail before the removal of values 4 and 6 as was represented on Figure . Then a `pop` restores the `size` to `9` (as it was at the time of the `push`) and the two removed values are reinserted in the set as expected. Notice on Figure 5 that the permutation of values is not the same as initially but it represents exactly the same set of values $\{0, \ldots, 8\}$. This works only because the set can only monotonically decrease its size between two push operations (therefore we do not allow both removal and insertion operations).

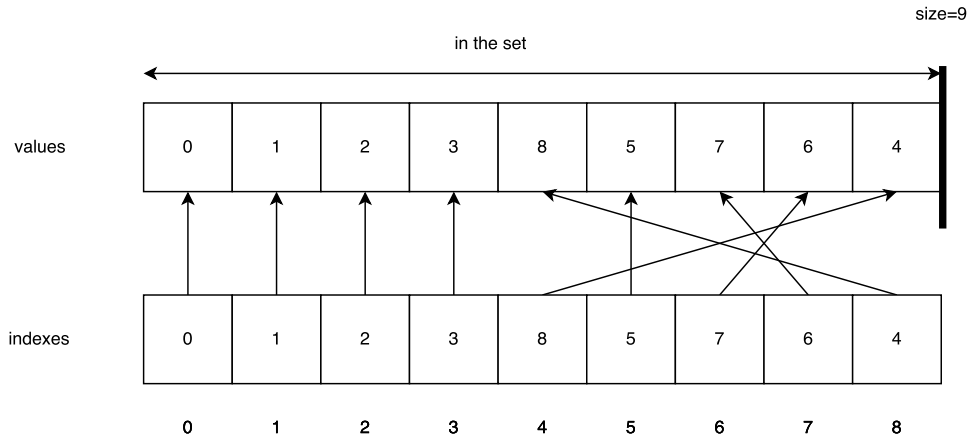

Fig. 5: Sparset Set

6.3 Blending Search and State Management

Once restorable states are available, it is easy to cleanly segregate the search and the state restoration logic. Indeed, one can simply inserts calls to create and restore batches of changes around the recursive call as shown in Listing 15.

Listing 15: DFS With Explicit State Handling

```
1
2 public class DFS {
3    private Trail  trail;
4    private Choice branching;
5    public DFS(Trail t,Choice b)  { trail = t; branching = b;}
6    public void dfs() {
7       Alternative[] alternatives = branching.call();
8       if (alternatives.length == 0)
9          notifySolution();
10      else
11         for (a : alternatives) {
12            trail.push();
13            a.call();
14            dfs();
15            trail.pop();
16         }
17   }
18 }
```

Writing an entire program that uses decision variables to produces, via backtracking, the tree shown in Figure 3 is somewhat direct and illustrated in Listing 16.

Listing 16: Generating a simple complete binary tree.

```
Solver cp = makeSolver();
BoolVar[] values = makeBoolVarArray(cp,3);

DFSearch search = new DFSearch(cp.getTrail(),() -> {
    int sel = -1;
    for (int i = 0 ; i < values.length;i++)
        if (values[i].getSize() > 1 && sel == -1)
            sel = i;
    final int i = sel;
    if (i == -1)
        return TRUE;
    else return branch(() -> equal(values[i],0),
                       () -> equal(values[i],1));
});

search.onSolution(() -> System.out.println(Arrays.toString(values))).dfs();
```

Note how the brancher passed to the `DFSSearch` constructor uses a simple loop to select an unbound variable and store its index in i[5]. If all variables are bound, the brancher returns `TRUE`: an empty array of alternatives. Otherwise, it creates a list with two constraints: $(values[i] = 0, values[i] = 1)$. The steps executed during the search at the creation of child nodes are described and illustrated on Figure 6. First the branching creates alternatives. Second, a batch receptacle is created on the trail with a call to `trail.push()`. Third, an alternative is executed. This can have side effects that modify part of the state embedded in variables or even constraints. All such modifications must occur under the auspices of reversible objects which will log with the trail the necessary undo operations. Finally, when the search backtracks, a call to `trail.pop()` restores the state by undoing all the operations tracked in the current batch and the execution can continue with the exploration of the next branch.

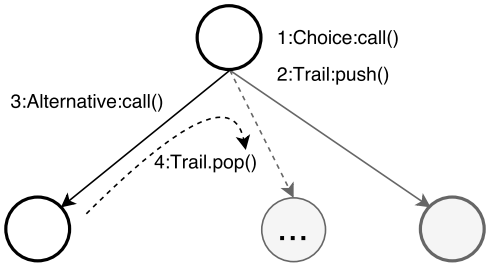

Fig. 6: Branching

6.4 Search and Propagation

As indicated earlier, a CP solver like MiniCP must fallback on *decomposition* by adding branching constraints when propagation alone cannot deliver a solution as reported by `success`(D). Conceptually, a simple decomposition can be $(x = v, x \neq v)$ for some constant $v \in D(x)$. While the specification calls for the filtering of the domain D_0 to obtain $D_1 = \mathcal{F}_{C \wedge x=c}(D_0)$ for the first sub-problem and $D_2 = \mathcal{F}_{C \wedge x \neq c}(D_0)$ for the second, in practice one can achieve the same result without creating and adding two new constraints to the constraint store C. Instead, one can simply do a direct modification of the state of $D(x)$, effectively doing $D(x) = \{v\}$ in the first case and $v \notin D(x)$ in the second. Since both are modification of the state they will cause the scheduling of *events* and the propagation will correctly compute the filtered domains. Yet, it is important to recall that the propagation could end with one of three outcomes:

1. `success`(D_1) in which case no further branching is needed and the recursive call to the `dfs` method will yield an empty branching and the invocation of `notifySolution`.

[5] The final annotation is a Java 8 idiosyncrasy as closures should not refer to a mutable value.

2. `failure`(D_1) in which case the sub-problem is infeasible and can be discarded. No further recursive calls to the `dfs` recursive method are needed.
3. $\neg$`success`(D_1) $\wedge\neg$`failure`(D_1) and a recursive call is warranted to further sub-divide the problem.

Since `failure` events are reported through exceptions in MiniCP, the `dfs` method should wrap the invocation of the alternative in a `try-catch` block to intercept any `InconsistencyException` and properly report the failure before moving on. Note how the recursive call to `dfs` is embedded into the `try` block as well so that it only happens on successes and further attempts at sub-division. This ultimate revision of the search code appears in Listing 17.

Listing 17: Depth First Search: Full Code.

```
1  public class DFS {
2     private Trail  trail;
3     private Choice branching;
4     public DFS(Trail t,Choice b)  { trail = t; branching = b;}
5     public void dfs() {
6        Alternative[] alternatives = branching.call();
7        if (alternatives.length == 0)
8           notifySolution();
9        else
10          for (a : alternatives) {
11             trail.push();
12             try {
13                a.call();
14                dfs();
15             } catch(InconsistencyException e) {
16                notifyFailure();
17             }
18             trail.pop();
19          }
20    }
21 }
```


6.5 Supporting Optimization

The resolution of a Constraint Optimization problems reduces to the resolution of 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 $\langle X, D, C, f\rangle$, first solve the CSP $\langle X, D, C\rangle$. As soon as a solution σ is found, evaluate $f(\sigma)$ to obtain a first upper bound f_1 on the objective function and solve the second CSP $\langle X, D, C \cup \{f < f_1\}\rangle$. Repeat this process until the k^{th} CSP $\langle X, D, C \cup \{f < f_1, f < f_2, \cdots, f < f_k\}$ in which the f_i's form a strictly monotonically decreasing sequence (i.e., $f_1 > f_2 > \cdots > f_k$) becomes infeasible. At that point, the last feasible CSP delivered the optimal value f_k for the objective and its feasible solution is a globally optimal solution to the COP.

The implementation tactic outlined above is thankfully straightforward to implement with MiniCP. The objective function is represented as a standard integer variable. Then it is sufficient to implement a single constraint that models $f < f_i$. This constraint maintains, as part of its state, the variable representing f, and the current best upper bound (f_i). When the constraint is posted, it performs three tasks:

1. Whenever the variable representing f (i.e., x in Listing 18) see its bounds change, it tightens the upper bound of x to remove every value $k \geq$ *bound*.
2. Whenever a solution is found, it updates the instance variable *bound* to reflect the tightening of the upper bound from f_i to $f_{i+1} < f_i$.
3. Whenever the event `failure`(D) occurs during the propagation of constraints, it schedules itself to make sure that the bound on the objective function will be tightened again.

Naturally, this outline implies that MiniCP provides notifications to execute arbitrary closures whenever a solution is found (`onSolution`) and whenever a failure is detected (`onFail`). Both can be achieved by augmenting the `dfs` method of the `DFSSearch` class to invoke a list of closures whenever such events are noted.

Listing 18: Minimize Constraint

```
1  public class Minimize extends Constraint {
```

```
    public int bound = Integer.MAX_VALUE;
    public final IntVar x;
    public final DFSearch dfs;
    public Minimize(IntVar x, DFSearch dfs) {
        super(x.getSolver());
        this.x = x;
        this.dfs = dfs;
    }
    protected void tighten() {
        if (!x.isBound()) throw new RuntimeException("objective not bound");
        this.bound = x.getMax() - 1;
    }
    @Override
    public void post() throws InconsistencyException {
        x.whenBoundsChange(() -> x.removeAbove(bound));
        dfs.onSolution(() -> {
            tighten();
            cp.schedule(this);
        });
        dfs.onFail(() -> cp.schedule(this));
    }
}
```

7 Advanced Techniques

7.1 Reified Constraints

A reified constraint $b \Leftrightarrow c$ is a constraint c that is a linked to a boolean b. It holds if and only if b is true, otherwise the negation of the constraint $\neg c$ must hold. The most frequent reified constraint is $b \Leftrightarrow c = c$ that we call the `IsEqual` constraint. The code for the `IsEqual` is given in Listing 19.

Listing 19: IsEqual

```
public class IsEqual extends Constraint { // b <=> x == c

    private final BoolVar b;
    private final IntVar x;
    private final int c;

    public IsEqual(BoolVar b, IntVar x, int c) {
        super(x.getSolver());
        this.b = b;
        this.x = x;
        this.c = c;
    }

    public void post() throws InconsistencyException {
        if (b.isTrue()) {
            x.assign(c);
        } else if (b.isFalse()) {
            x.remove(c);
        } else if (x.isBound()) {
            b.assign(x.getMin() == c);
        } else if (!x.contains(c)) {
            b.assign(0);
        } else {
            b.whenBind(() -> {
                if (b.isTrue()) x.assign(c);
                else x.remove(c);
            });
            x.whenBind(() ->
                b.assign(x.getMin() == c)
            );
            x.whenDomainChange(() -> {
                if (!x.contains(c))
                    b.assign(0);
            });
        }
    }
}
```

7.2 Views

7.3 Limits

7.4 Large Neighborhood Search

Listing 20: Large Neighborhood Search

```
dfs.onSolution(() -> {
    // Update the current best solution
    for (int i = 0; i < n; i++) {
        xBest[i] = x[i].getMin();
    }
    System.out.println("objective:"+objective.getMin());
});
for (int i = 0; i < nRestarts; i++) {
    System.out.println("restart number #"+i);

    // Record the state such that the fragment constraints can be cancelled
    cp.push();

    // Assign the fragment 5% of the variables randomly chosen
    for (int j = 0; j < n; j++) {
        if (rand.nextInt(100) < 5) {
            equal(x[j],xBest[j]);
        }
    }
    dfs.start(statistics -> statistics.nFailures >= failureLimit);

    // cancel all the fragment constraints
    cp.pop();
}
```

8 Suggested Syllabus and lectures

9 Conclusion

References

1. Boussemart, F., Lecoutre, C., Piette, C.: Xcsp3: An integrated format for benchmarking combinatorial constrained problems. arXiv preprint arXiv:1611.03398 (2016)
2. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog user's manual, vol. 3. Swedish Institute of Computer Science Kista, Sweden (1988)
3. Dynadec, Van Hentenryck, P., Michel, L., Schaus, P.: Comet v2. 1 user manual (2009)
4. Floyd, R.W.: Nondeterministic algorithms. Journal of the ACM (JACM) **14**(4), 636–644 (1967)
5. Knuth, D.E.: The art of computer programming: Volume 4B, Combinatorial Algorithms: Part 2, Backtrack Programming, vol. 4B. Adison-Wesley (2016)
6. Kuchcinski, K., Szymanek, R.: Jacop-java constraint programming solver. Procs. of CP Solvers: Modeling, Applications, Integration, and Standardization (2013)
7. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. Constraints pp. 1–45 (2014)
8. van Omme, N., Perron, L., Furnon, V.: or-tools users manual. Tech. rep., Technical report, Google (2014)
9. OscaR Team: OscaR: Scala in OR (2012). Available from `https://bitbucket.org/oscarlib/oscar`
10. Prudhomme, C., Fages, J.G., Lorca, X.: Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR **6241** (2014)
11. de Saint-Marcq, V.l.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS), pp. 1–10 (2013)
12. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. Handbook of Constraint Programming p. 493 (2006)
13. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. Foundations of Artificial Intelligence **2**, 495–526 (2006)
14. Stuckey, P.: Modeling discrete optimization. `https://www.coursera.org/learn/modeling-discrete-optimization` (2015)
15. 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)
16. 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)
17. 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)