MaxiCP: A Not So Mini Constraint Programming Solver

Pierre Schaus [1], Guillaume Derval [2], Augustin Delecluse [3], Laurent Michel [4], and Pascal Van Hentenryck [5]

[1]UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium
[2]ULiège, Montefiore Institute, Liège, Belgium
[3]KU Leuven, Leuven, Belgium
[4]University of Connecticut, Storrs, USA
[5]Georgia Institute of Technology, Atlanta, USA

Constraint Programming (CP) is a powerful paradigm for solving complex combinatorial optimization problems. This paper presents MaxiCP, a modern, high-performance, open-source CP solver designed to bridge the gap between educational simplicity and industrial-grade efficiency. Building upon MiniCP [1] and incorporating ideas from the OscaR solver [2]and Objective-CP [3], MaxiCP [4] introduces several novelties: (1) conditional time-interval variables and cumulative functions for scheduling, implemented through the Generalized Cumulative constraint [5]. (2) sequence variables' computational domain for routing, supporting insertion-based search heuristics and Large Neighborhood Search [6] and (3) a symbolic modeling layer (MaxiCP-Modelling) that treats models as first-class immutable objects, enabling model transformations, reformulations, and parallel resolution strategies [7]; We provide a detailed description of the solver's architecture, including its state management system based on trailing, its event-driven propagation engine, and its two-layer API design. Through a series of models, we show the expressive power of MaxiCP. All source code of MaxiCP and models are available as an open-source Java library under the MIT license.

1 Introduction

Constraint Programming (CP) provides a declarative framework where combinatorial problems are modeled in terms of variables and constraints. The resolution process combines exhaustive search with filtering algorithms (propagators) that prune the search space by removing inconsistent values from variable domains. The paradigm $CP = Model + Search$ captures its essence: a user writes a model, and the solver searches for solutions, guided by customizable search heuristics.

CP has been successfully applied to a wide range of industrial applications, from scheduling [8] and routing [9] to configuration, timetabling, and resource allocation. Commercial solvers such as IBM's CP Optimizer [8] or OptalCP [10] provides high performance and rich modeling abstractions, but their proprietary nature restricts access to their internal implementations, which limits their use in research and education where source code access is often required for experimentation and extension. Open-source solvers, on the other hand, vary widely in

completeness and documentation. Notable open-source solvers are Choco [11], Gecode [12] and OR-Tools [13] with a very rich documentation, that are also high quality solvers that have been maintained over the years.

The architecture of MaxiCP builds upon MiniCP [1], which was originally designed as a lightweight, educational solver to illustrate the core principles of constraint programming. Its elegant and minimalist design offers a near one-to-one correspondence between theoretical CP concepts and their implementation, making it an excellent pedagogical tool. However, MiniCP is not intended for industrial-scale applications. It lacks several features required for competitive performance such as rich computational domains, efficient search strategies, and its constraint library remains limited. Additionally, some parts of the implementation are deliberately not fully optimized, as the primary goal is to prioritize code readability and simplicity.

This does not mean that MaxiCP sacrifices readability—its codebase also remains clear and well-structured. It incorporates algorithmic ideas from the OscaR [2] and Objective-CP [3] CP solvers such as priority-based propagation queues, incremental constraint computations based on advanced mechanisms such as delta changes of the domains [14]. It also includes novel functionalities such as insertion-based sequence variables [6, 6, 15], some of them highly inspired from commercial solvers such as conditional task intervals [16, 17]. MaxiCP also proposes some efficient black-box search strategies such as Conflict Ordering Search [18] or Failure Directed Search [19]. The result is a solver that is simultaneously well-suited for education, research, and practical deployment.

Like its predecessor MiniCP, MaxiCP is implemented in Java[1] similarly to other successful CP solvers (e.g. Choco [11], or ACE solver [20]).

Scope of this document. This document is not intended as an introduction to Constraint Programming. Readers seeking a conceptual overview of CP—including domains, propagation, and search—are referred to the MiniCP paper [1] and related literature (e.g. [21]). Instead, this document illustrates the main functionalities of MaxiCPto model and solve combinatorial optimization problems.

More specifically, the document:

- presents the main modeling and solving capabilities of MaxiCP, illustrated through representative code examples;
- demonstrates how common problem classes—including combinatorial optimization, scheduling, and vehicle routing—can be expressed using appropriate variables, constraints, and search strategies;
- describes the two-level design of the API, distinguishing between the low-level engine interface and the higher-level symbolic modeling layer.

The primary functionalities of MaxiCP discussed in this document are:

- **Basic Functionalities**: This includes the state-management and the trail, implementing a custom search, implementing a constraint, and solving a problem with Large Neighborhood Search (LNS) [22].
- **Advanced Scheduling with Conditional Time Intervals**: Full support for conditional time-interval variables and cumulative function expressions, as introduced in [17] and implemented through the Generalized Cumulative constraint [5]. This enables the natural modeling of optional tasks, alternative resources, and producer-consumer scheduling problems.

[1] Java offers an excellent trade-off between ease of development, code stability, ease of use, and execution speed. Its strong type system, automatic memory management, and mature tooling ecosystem make it well suited for building complex combinatorial solvers that remain maintainable over time. Moreover, recent advances in the Java ecosystem, such as ahead-of-time compilation to native code (e.g., via GraalVM), further improve performance and deployment flexibility.

- **Sequence Variables (`SeqVar`)**: A dedicated computational domain for routing and sequencing problems [6]. Sequence variables represent partial paths through node insertions rather than successor arrays, natively supporting optional visits, insertion-based search heuristics, and Large Neighborhood Search. Dedicated global constraints such as `distance`, `transitionTimes`, and `cumulative` are provided for vehicle routing.
- **Symbolic Modeling (MaxiCP-Modelling)**: A modeling layer that treats models as immutable symbolic linked lists of constraints [7]. This design enables model transformations and reformulations, the creation of sub-problems and neighborhoods for Large Neighborhood Search, and embarrassingly parallel search where multiple solver instances work on the same immutable model.

The remainder of this paper is structured as follows. Section 2 gives an overview of the architecture, introduces the package organization, and presents the two-layer API (raw and symbolic) through a side-by-side comparison on the N-Queens problem. From Section 3 onward, we adopt a bottom-up presentation, starting with the implementation layer: Section 3 covers state management and trailing, Section 4 details the propagation engine, Section 5 covers the search framework including Large Neighborhood Search (LNS) [22] and Variable Objective Large Neighborhood Search (VOLNS) [23]. We then turn to advanced variable domains: Section 6 covers conditional time-interval variables and scheduling, Section 7 presents sequence variables for routing, We finish by introducing the symbolic modeling in Section 8 and its support for model transformations, reformulations, and parallel search. Section 9 concludes.

2 Architecture Overview

MaxiCP is implemented in Java (version 17+) and consists of approximately 40 000 lines of code. Its architecture is organized into several coherent packages:

`org.maxicp.state` provides the state management primitives and data-structures used for backtracking and the *trailing* default implementation.

`org.maxicp.cp.engine.core` contains the core CP engine: the solver (`MaxiCP`), variable implementations (`CPIntVar`, `CPBoolVar`, `CPSeqVar`, `CPIntervalVar`), domains, and the priority-based propagation queue.

`org.maxicp.cp.engine.constraints` contains the constraint implementations, organized into sub-packages for some specific computation domains.

`org.maxicp.modeling` provides the symbolic modeling layer, including symbolic variables (`IntVar`, `SeqVar`, `IntervalVar`), the `Factory` with constraint creation methods, and the symbolic model representation.

`org.maxicp.cp.modeling` bridges the symbolic modeling layer and the concrete CP layers. The `ConcreteCPModel` class instantiates symbolic constraints into the CP engine.

`org.maxicp.search` implements search strategies, including depth-first search (`DFSearch`), concurrent search (`ConcurrentDFSearch`), best-first search (`BestFirstSearch`), and a rich library of variable and value selection heuristics (`Searches`).

2.1 Two Levels of Modeling: Raw and Symbolic

A distinguishing feature of MaxiCP is that it offers *two* ways to model and solve problems:

1. **The raw (engine-level) API.** The user creates a `CPSolver` instance and works directly with concrete variable objects (`CPIntVar`, `CPSeqVar`, `CPIntervalVar`). Constraints are posted on the solver via `cp.post(...)`. Branching closures modify the solver state by posting equality/disequality constraints. This API gives maximal control over the engine and is the natural choice when implementing custom constraints, propagators, or search heuristics. It mirrors the API of MiniCP, so users familiar with MiniCP will feel immediately at home.

2. **The symbolic (modeling-level) API.** The user creates a `ModelDispatcher` and works with symbolic variable objects (`IntVar`, `SeqVar`, `IntervalVar`). Constraints are added via `model.add(...)`. The model is an immutable linked list; adding a constraint returns a new model node, leaving the original unchanged. The model must be *concretized* into a solver engine before it can be solved. This indirection enables model transformations, LNS neighborhoods as model branches, and embarrassingly parallel search. The symbolic layer is described in detail in Section 8.

The two APIs are remarkably similar in surface syntax. To illustrate this, Listings 1 and 2 show the NQueens problem implemented in each style.

Listing 1: N-Queens with the raw API (`CPSolver`)

```
int n = 8;
CPSolver cp = CPFactory.makeSolver();
CPIntVar[] q  = CPFactory.makeIntVarArray(cp, n, n);
CPIntVar[] qL = CPFactory.makeIntVarArray(n, i -> minus(q[i], i));
CPIntVar[] qR = CPFactory.makeIntVarArray(n, i -> plus(q[i], i));

cp.post(allDifferent(q));
cp.post(allDifferent(qL));
cp.post(allDifferent(qR));

DFSearch search = CPFactory.makeDfs(cp, () -> {
    CPIntVar qs = selectMin(q,
            qi -> qi.size() > 1,
            qi -> qi.size());
    if (qs == null) return EMPTY;
    int v = qs.min();
    return branch(
        () -> cp.post(eq(qs, v)),
        () -> cp.post(neq(qs, v)));
});

search.onSolution(() ->
    System.out.println(Arrays.toString(q)));
SearchStatistics stats = search.solve();
```

Listing 2: N-Queens with the symbolic API (`ModelDispatcher`)

```
int n = 12;
ModelDispatcher model = makeModelDispatcher();
IntVar[] q  = model.intVarArray(n, n);
IntExpression[] qL = model.intVarArray(n, i -> q[i].plus(i));
IntExpression[] qR = model.intVarArray(n, i -> q[i].minus(i));

model.add(allDifferent(q));
model.add(allDifferent(qL));
model.add(allDifferent(qR));

Supplier<Runnable[]> branching = () -> {
    IntExpression qs = selectMin(q,
            qi -> qi.size() > 1,
            qi -> qi.size());
    if (qs == null) return EMPTY;
    int v = qs.min();
    return branch(
        () -> model.add(eq(qs, v)),
        () -> model.add(neq(qs, v)));
};

ConcreteCPModel cp = model.cpInstantiate();
DFSearch dfs = cp.dfSearch(branching);
dfs.onSolution(() ->
    System.out.println(Arrays.toString(q)));
SearchStatistics stats = dfs.solve();
```

Both APIs share the same search infrastructure (`DFSearch`, `SearchStatistics`, `Searches`), the same heuristics (`firstFailBinary`, `setTimes`, etc.), and the same constraint library. The branching function in the NQueens examples defines a custom search strategy where the solver selects the variable with the smallest domain (first-fail heuristic) and branches by assigning its minimum value or removing it. While writing custom branching closures provides complete control over the search, as detailed in Section 5, MaxiCP also provides a variety of pre-built heuristics and combinators that simplify search definition. In the following sections we adopt a bottom-up presentation, starting from the implementation of the (raw) layer main objects: state management, propagation, and search. The symbolic modeling layer is covered in depth in Section 8.

3 State Management

CP solvers must efficiently save and restore states during search, which is inherently organized as a depth-first search (DFS). In DFS-based constraint programming, the solver explores a decision tree by making a branching decision, recursively continuing the search, and backtracking when a failure or dead-end is reached. This execution model requires a mechanism to restore a previous consistent state before exploring an alternative branch. MaxiCP is a *trail-based* solver: rather than saving the entire state at each choice point, it relies on a stack-based memory management scheme called the *trail*. We refer to [1] for a comprehensive introduction to trailing and its implementation in CP solvers. Very briefly, the trail records only the incremental changes that occur during propagation and search. This makes it possible to efficiently undo modifications when backtracking in the DFS tree. The trail mechanism is available through the `StateManager` interface that exposes the following key operations:

Listing 3: StateManager interface

```
public interface StateManager {
    void saveState();       // Push a new save point
    void restoreState();    // Restore to last save point
    int getLevel();         // Current depth in the tree

    StateInt makeStateInt(int initValue);
    <T> State<T> makeStateRef(T initValue);
    <K,V> StateMap<K,V> makeStateMap();
}
```

All stateful components of the solver—variable domains, propagation queues, the depth first search and even the symbolic model reference—are built on top of these primitives. For instance, the domain of a `CPIntVar` is backed by `StateInt` (i.e. reversible integers) for its minimum and maximum bounds, and by a reversible sparse set for the set of values in the domain [1, 14].

Usage Example. Listing 4 illustrates the use of trailing:

Listing 4: Trailing example in MaxiCP

```
StateManager sm = new Trailer();
StateInt counter = sm.makeStateInt(0);

sm.saveState();           // Level 0 -> Level 1
counter.setValue(10);     // Trail records: (counter, 0)

sm.saveState();           // Level 1 -> Level 2
counter.setValue(42);     // Trail records: (counter, 10)

sm.restoreState();        // Level 2 -> Level 1, counter = 10
sm.restoreState();        // Level 1 -> Level 0, counter = 0
```

The trailing mechanism is implemented in the `Trailer` class. Conceptually, every time the solver descends in the search tree, it creates a new decision level by calling the `saveState()` method, which pushes a new save point onto the trail stack. Any modification performed at that level is recorded in the trail so that it can be undone when the solver returns to a previous level on backtrack.

4 Propagation Engine

The propagation engine is the heart of any CP solver. In MaxiCP, it is implemented in the `MaxiCP` class and follows a priority-based fixed-point algorithm.

4.1 Priority-Based Scheduling

Constraints in MaxiCP implement the `CPConstraint` interface, which defines a `propagate()` method and a `priority()` method. When a variable domain changes, the constraints that are subscribed to that event are *scheduled* for propagation. The engine processes constraints from a priority queue, ensuring that lightweight, fast-propagating constraints (e.g., unary constraints, equality) are executed before expensive global constraints (e.g., `AllDifferent`, `Cumulative`).

Listing 5: The fixed-point propagation loop in MaxiCP

```
public void fixPoint() {
    notifyFixPoint();
    while (!propagationQueue.isEmpty()) {
        CPConstraint c = propagationQueue.poll();
        c.setScheduled(false);
        if (c.isActive())
            c.propagate();
    }
}

public void schedule(CPConstraint c) {
    if (c.isActive() && !c.isScheduled()) {
        c.setScheduled(true);
        propagationQueue.add(c, c.priority());
    }
}
```

If a constraint's propagation triggers an `InconsistencyException`, the queue is cleared and the exception propagates up to the search layer, which handles backtracking.

4.2 Event-Driven Constraint Activation

Constraints register themselves to specific domain events. When a variable's domain is modified, only the relevant constraints are re-scheduled. The available events depend on the variable type:

- **Integer variables (`CPIntVar`):**
  - `propagateOnFix`: triggered when the variable is assigned a single value.
  - `propagateOnBoundChange`: triggered when the minimum or maximum of the domain changes.
  - `propagateOnDomainChange`: triggered when any value is removed from the domain.
- **Interval variables (`CPIntervalVar`):**
  - `propagateOnChange`: triggered when any attribute of the interval (start, end, length, presence) changes.
  - `propagateOnFix`: triggered when the interval is fully fixed.
- **Sequence variables (`CPSeqVar`):**
  - `propagateOnInsert`: triggered when a node is inserted into the partial sequence.
  - `propagateOnExclude`: triggered when a node is excluded from the sequence.
  - `propagateOnRequire`: triggered when a node becomes required.

This fine-grained event system avoids unnecessary re-propagation and is critical for the solver's performance. Notice that registering a constraint to an event is a reversible operation: it becomes part of the current search state and will automatically be undone upon backtracking thanks to the trail-based state management.

4.3 Posting Constraints

Constraints are posted to the solver using `cp.post(constraint)`. Due to the reversibility of constraint registration, posting a constraint is also a reversible operation: it becomes part of the current search state and will automatically be undone upon backtracking thanks to the trail-based state management. This property is important in a depth-first search setting, where alternative branches are explored by temporarily extending the current state rather than copying it.

This method invokes the constraint's `post()` method, which typically:

1. Registers the constraint to the relevant variable events.
2. Performs an initial propagation.

By default, posting a constraint immediately triggers a full fixed-point computation, ensuring that all propagation consequences of the new constraint are fully enforced before search continues. This behavior is precisely why constraint posting is well-suited for branching: search decisions can be implemented by posting constraints that define each alternative branch, letting propagation automatically explore their consequences.

This propagation step can be deferred by passing `false` as a second argument, i.e., `cp.post(constraint, false)`. The fix-point can then be explicitly triggered later by calling `cp.fixPoint()`.

When posting a constraint, the constraint can fail immediately if it detects an inconsistency (e.g., an empty domain). In this case, an `InconsistencyException` is thrown, which is caught by the search engine to trigger backtracking.

4.4 Implementing a Custom Constraint

Implementing a custom constraint in MaxiCP involves extending the `AbstractConstraint` class and implementing the `post()` and `propagate()` methods. Listing 6 shows the implementation of a $x \leq y$ constraint.

Listing 6: Implementation of the LessOrEqual constraint

```
public class LessOrEqual extends AbstractConstraint {
    private final CPIntVar x, y;

    public LessOrEqual(CPIntVar x, CPIntVar 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());
    }
}
```

The `post()` method is responsible for registering the constraint to variable events. In this case, the constraint is re-scheduled whenever the upper bound of x or the lower bound of y changes. The `propagate()` method then prunes the domains. Whenever a domain becomes empty, through the call to `removeAbove` or `removeBelow`, an `InconsistencyException` is thrown, which signals the search engine to backtrack. For some constraint, it may be necessary to explicitly throw an `InconsistencyException` when a failure is detected.

4.5 Implementing a Custom Constraint with Delta

A more efficient way to implement propagators is to take advantage of *delta events*, which allow the constraint to react only to changes in variable domains rather than re-scanning full domains at each propagation. This is explained in [14].

Listing 7 and 8 show a possible implementation of a `Element` constraint, which enforces $z = t[y]$ using delta-based incremental propagation.

The constraint maintains two-directional consistency between y and z:

- A value $v \in \mathrm{dom}(y)$ is *supported* if and only if $t[v] \in \mathrm{dom}(z)$. When a value is removed from $\mathrm{dom}(z)$, every index v in $\mathrm{dom}(y)$ with $t[v] = v$ loses its support and must be removed.
- A value $w \in \mathrm{dom}(z)$ is *supported* if and only as there exists at least one index $v \in \mathrm{dom}(y)$ such that $t[v] = w$. When that count drops to zero, w can no longer be achieved and must be removed from $\mathrm{dom}(z)$.

The key data structure is `supportCounter`, an array of *reversible integers* (i.e. `StateInt` objects backed by the trail). Its size equals the initial size of $\mathrm{dom}(z)$, and it is indexed by `v - offZ`, where `offZ = z.min()` is the offset recorded at posting time.

$$\texttt{supportCounter}[w - \texttt{offZ}] \;=\; \left|\{\, v \in \mathrm{dom}(y) \mid t[v] = w \,\}\right|$$

In words, `supportCounter[w - offZ]` counts the number of indices currently in $\mathrm{dom}(y)$ that map to value w through the array t. This count is computed once during `post()` and maintained incrementally during `propagate()`:

- When a value v is removed from $\mathrm{dom}(y)$ (detected via `yDelta`), the counter for $t[v]$ is *decremented*. If it reaches 0, no index in $\mathrm{dom}(y)$ maps to $t[v]$ any more, so $t[v]$ is removed from $\mathrm{dom}(z)$.
- When a value w is removed from $\mathrm{dom}(z)$ (detected via `zDelta`), every index $v \in \mathrm{dom}(y)$ with $t[v] = w$ is removed from $\mathrm{dom}(y)$. Note that these removals will in turn trigger further decrements of the affected counters, but since w is already gone from $\mathrm{dom}(z)$, those counters reaching 0 require no further action.

Because `supportCounter` elements are `StateInt` objects, their values are automatically restored upon backtracking, keeping the counters consistent with the current search state at every node of the search tree.

The objects `yDelta` and `zDelta` are created by `y.delta(this)` and `z.delta(this)` respectively. A delta tracks, for the constraint which values have been removed from the variable's domain *since the last time this constraint was propagated*. Calling `yDelta.fillArray(yValues)` fills the array with those recently removed values and returns their count. This avoids iterating over the entire remaining domain and is the key mechanism that makes the propagator incremental.

Listing 7: Delta-based implementation of Element (part 1)

```
public class Element extends AbstractCPConstraint {

    private final int[] t;
    private final CPIntVar y, z;
    private DeltaCPIntVar yDelta, zDelta;
    private int[] yValues, zValues;
    private int offZ;
    private StateInt[] supportCounter;

    public Element(int[] array, CPIntVar y, CPIntVar z) {
        super(y.getSolver());
        this.t = array;
        this.y = y;
        this.z = z;
    }

    @Override
    public void post() {
        y.removeBelow(0);
        y.removeAbove(t.length - 1);
        y.propagateOnDomainChange(this);
        yDelta = y.delta(this);
        yValues = new int[y.size()];

        z.removeBelow(Arrays.stream(t).min().getAsInt());
        z.removeAbove(Arrays.stream(t).max().getAsInt());
        z.propagateOnDomainChange(this);
        zDelta = z.delta(this);
        zValues = new int[z.size()];

        int sizeY = y.fillArray(yValues);
        for (int i = 0; i < sizeY; i++) {
            int v = yValues[i];
            if (!z.contains(t[v])) {
                y.remove(v);
            }
        }
        supportCounter = new StateInt[z.size()];
        offZ = z.min();
        for (int i = 0; i < z.size(); i++) {
            supportCounter[i] =
            getSolver().getStateManager().makeStateInt(0);
        }
        for (int i = 0; i < t.length; i++) {
            if (y.contains(i)) {
                supportCounter[t[i] - offZ].increment();
            }
        }
        int sizeZ = z.fillArray(zValues);
        for (int i = 0; i < sizeZ; i++) {
            int v = zValues[i];
            if (supportCounter[v - offZ].value() == 0) {
                z.remove(v);
            }
        }
        propagate();
    }
}
```

Listing 8: Delta-based implementation of Element (part 2)

```
@Override
public void propagate() {

    if (zDelta.size() > 0) {
        int size = zDelta.fillArray(zValues);
        for (int i = 0; i < size; i++) {
            int v = zValues[i];

            int sizeY = y.fillArray(yValues);
            for (int j = 0; j < sizeY; j++) {
                int yv = yValues[j];
                if (t[yv] == v) {
                    y.remove(yv);
                }
            }
        }
    }

    if (yDelta.size() > 0) {
        int size = yDelta.fillArray(yValues);
        for (int i = 0; i < size; i++) {
            int v = yValues[i];

            StateInt counter = supportCounter[t[v] - offZ];
            if (counter.decrement() == 0) {
                z.remove(t[v]);
            }
        }
    }
}
```

4.6 Global Constraints in MaxiCP

This subsection lists some important global constraints available in MaxiCP, with their semantics and references on their filtering algorithms.

Table constraint (`TableCT`). Given variables $x = \langle x_1, \ldots, x_n \rangle$ and a table $T \subseteq \mathbb{Z}^n$, the constraint enforces that the assignment of x matches one row of T:

$$\exists r \in T : \ \forall j \in \{1, \ldots, n\}, \ x_j = r_j.$$

This is the classical extensional (in-extension) constraint. MaxiCP relies the compact-table filtering [24].

Bin packing constraint (`BinPacking`). Given item-to-bin variables x_i, item weights w_i, and bin-load variables $load_b$, the constraint enforces exact bin loads:

$$\forall b : \ load_b = \sum_{i | x_i = b} w_i.$$

The filtering of relies on [25–27].

Global cardinality constraint (GCC). For value set V and occurrence bounds, GCC constrains the number of variables taking each value. With upper bounds only (as in `CostCardinalityMaxDC` without costs), semantics is:

$$\forall v \in V : \ |\{i \mid x_i = v\}| \leq U_v.$$

The arc-consistent filtering is due to Régin [28] but MaxiCP relies on the algorithm of [29].

Soft cardinality constraint (`SoftCardinalityDC`). For values v with lower/upper bounds L_v, U_v, let $c_v = |\{i \mid x_i = v\}|$. The violation per value is

$$\text{viol}(v) = \max(0, L_v - c_v, c_v - U_v),$$

and the global violation variable is constrained by

$$viol = \sum_v \text{viol}(v).$$

This constraint was introduced in [30] but the MaxiCP filtering relies on the one of [31, 32].

Cardinality with costs (`CostCardinalityMaxDC`). With assignment costs $cost_{i,v}$ and budget variable H, the constraint combines (upper-bounded) cardinality and a global cost limit:

$$\forall v : \ |\{i \mid x_i = v\}| \leq U_v, \qquad \sum_i cost_{i,x_i} \leq H.$$

The filtering algorithms for GCC with costs implemented in MaxiCP is the one of [29].

5 Search

MaxiCP provides a flexible, compositional search framework. The core search engine is a depth-first search with branch-and-bound for optimization.

5.1 Depth-First Search with Closures

Following the design of MiniCP [1], the search in MaxiCP is defined by a *branching function*: a supplier of an array of closures (the `Runnable` functional interface in Java), one for each child branch. When the array is empty, the current node is considered as a solution. Listing 9 illustrates this concept on the N-Queens problem:

Listing 9: Branching as a supplier of closures (raw API)

```
DFSearch search = makeDfs(cp, () -> {
    CPIntVar qs = selectMin(q, // first-fail: select variable
        qi -> qi.size() > 1, //   that is not fixed
        qi -> qi.size()); //   with smallest domain
    if (qs == null) return EMPTY; // all fixed -> solution
    int v = qs.min();
    return branch(
        () -> cp.post(eq(qs, v)), // left: assign v
        () -> cp.post(neq(qs, v)) // right: remove v
    );
});
```

The `branch` method returns an array of runnables. The `selectMin` utility selects the variable with the smallest domain among unfixed variables, implementing the well-known *first-fail*

heuristic. Each branch closure calls `cp.post()` to add a constraint. The posting of the constraint is thus not immediately executed when the branching function is called, but rather deferred until the corresponding branch is explored in the search tree. As described in Section 4, `post` registers the constraint and triggers a full fix-point computation, propagating all consequences of the branching decision. If the fix-point detects an inconsistency (an empty domain), an `InconsistencyException` is thrown and the search backtracks.

5.2 Built-In Heuristics: Variable and Value Selectors

Writing a branching closure from scratch, as in Listing 9, gives full control but is verbose. The `Searches` class offers a compositional alternative by separating two orthogonal concerns:

1. A *variable selector* (`Supplier<IntExpression>`): returns the next unfixed variable to branch on, or `null` when all variables are fixed.
2. A *value selector* (`Function<IntExpression, Integer>`): given the selected variable, returns the value to try first.

These selectors are combined by the factory methods `heuristicBinary` and `heuristicNary` to produce complete branching strategies, as illustrated next on the N-Queens problem.

Binary branching. `heuristicBinary(varSel)` produces a binary branching: at each node, the variable selector picks a variable x; the left branch assigns $x = \min(\mathrm{dom}(x))$ and the right branch removes that value ($x \neq \min(\mathrm{dom}(x))$). An overload `heuristicBinary(varSel, valSel)` lets the user supply a custom value selector instead of the default minimum. Listing 10 shows three equivalent ways to write a first-fail search for NQueens.

Listing 10: N-Queens: three ways to express first-fail binary search

```
// (a) most concise: built-in convenience method
DFSearch s1 = makeDfs(cp, firstFailBinary(q));

// (b) explicit variable selector, default value (min)
DFSearch s2 = makeDfs(cp,
    heuristicBinary(minDomVariableSelector(q)));

// (c) explicit variable and value selectors
DFSearch s3 = makeDfs(cp,
    heuristicBinary(minDomVariableSelector(q),
                    x -> x.min())); // value = min of domain
```

All three produce the same search tree. Variant (c) is the most flexible: the value selector can be replaced by any function, e.g. `x -> (x.min() + x.max()) / 2` to branch on the domain midpoint.

N-ary branching. `heuristicNary(varSel)` creates one child branch per value in the domain (in increasing order). An overload accepts a value heuristic $h : \mathbb{Z} \to \mathbb{Z}$ and sorts the domain values by increasing $h(v)$ before creating the branches.

Listing 11: N-Queens: n-ary branching with custom value ordering

```
// n-ary, values tried in increasing order
DFSearch s4 = makeDfs(cp,
    heuristicNary(minDomVariableSelector(q)));

// n-ary: values closer to n/2 tried first
DFSearch s5 = makeDfs(cp,
    heuristicNary(minDomVariableSelector(q),
                  v -> Math.abs(v - n / 2)));
```

Built-in variable selectors. Two variable selectors cover the most common needs:

- `minDomVariableSelector(x)`: first-fail—returns the unfixed variable with the smallest domain.
- `staticOrderVariableSelector(x)`: returns the first unfixed variable in the array order.

Custom variable selectors are easy to write; for instance, to select the variable with the largest domain:

Listing 12: Custom variable selector: largest domain first

```
Supplier<IntExpression> maxDom = () ->
    selectMin(q, qi -> !qi.isFixed(), qi -> -qi.size());
DFSearch s6 = makeDfs(cp, heuristicBinary(maxDom));
```

Advanced value selection. The method `boundImpactValueSelector(objective)` [33] implements the Bound-Impact strategy: for each candidate value of the selected variable, the solver temporarily assigns it, measures the resulting lower bound of the objective, then backtracks. The value that tightens the bound the least is returned, biasing the search toward balanced, high-quality subtrees.

Conflict-aware heuristics. Two conflict-based strategies wrap the variable/value selector mechanism:

- `lastConflict` [34]: after a failure, the variable that caused it is re-tried *before* consulting the fallback variable selector.
- `conflictOrderingSearch` [18]: maintains a conflict counter per variable; the variable with the highest count is selected first, falling back to the base selector when no conflicting variable remains.

Scheduling and sequencing heuristics. For interval variables:

- `fds(tasks)` implements the *Failure Directed Search* [19]. It is very effective for proving optimality and supports optional tasks. This search strategy makes branching decisions on the status, the start time, and the duration of the tasks.
- `setTimes(tasks)` [35] branches on the start time of the task with the smallest slack. It assumes all tasks are mandatory. This search only assigns start times and does not fix the status of the tasks, so it can still be applied when tasks are optional. However, it may be incomplete in cases where delaying a task is beneficial, such as in just-in-time scheduling.

- `rank(tasks)` imposes a total order on the tasks. It also assumes that all tasks are mandatory.

For sequence variables: `firstFailBinary(seqVars)` selects the insertable node with the fewest insertion points and creates one branch per possible predecessor (plus an exclusion branch for optional nodes); `branchesInsertingNode` sorts insertion branches by a user-supplied detour cost.

Combinators. `and(b1, b2, ...)` composes branching strategies sequentially: a branching in the list is activated only when all preceding ones return `EMPTY`. This is useful for instance in scheduling, where one first decides presence/absence and then fixes start times. `limitedDiscrepancy(branching, maxDisc)` wraps a branching and prunes any path that accumulates more than `maxDisc` right branches, implementing Limited Discrepancy Search [36].

5.3 Branch-and-Bound Optimization

To optimize, the user creates an `Objective` and calls `search.optimize(objective)`. The solver uses a branch-and-bound strategy: each time a solution is found, the objective bound is tightened to exclude worse solutions. Optimization also supports a limit closure, which is evaluated at each node to stop the search based on custom criteria, such as time or number of nodes explored: The criteria takes the current search statistics as input that contains information such as the number of nodes explored, the number of solutions found, and the elapsed time in milliseconds.

Listing 13: Optimization with a time limit

```
SearchStatistics stats = search.optimize(minimize(cost),
    stats -> stats.timeInMillis() > 60000);  // 60s limit
```

5.4 Large Neighborhood Search

Large Neighborhood Search (LNS) [22] is an effective meta-heuristic that leverages the strengths of CP on large-scale optimization problems. Its key idea is to improve an incumbent solution by *fixing* a large subset of the decision variables to their current best values while *relaxing* the remaining variables, then searching for an improving solution in that restricted subspace. The subspace exploration (often called the *repair* or *reconstruction* phase) is typically subject to a failure or time limit to avoid being stuck in unpromising regions. This process is repeated across many *restarts*, using randomization to select different subsets of variables to relax at each iteration.

We illustrate LNS on the Quadratic Assignment Problem (QAP). Given n facilities and n locations, a weight matrix w (flow between each pair of facilities) and a distance matrix d (distance between each pair of locations), the goal is to assign every facility to a distinct location so as to minimize $\sum_{i,j} w_{i,j} \cdot d_{x_i,x_j}$, where x_i is the location assigned to facility i.

Listing 14 shows the model. An array of integer variables `x[i]`$\in \{0, \ldots, n-1\}$ represents the location of each facility. The `allDifferent` constraint ensures that no two facilities share a location. The objective is expressed by using the `element` constraint: `element(d, x[i], x[j])` returns the variable $d[x_i][x_j]$, i.e., the distance between the locations assigned to facilities i and j. Multiplying by the corresponding weight $w_{i,j}$ and summing over all pairs yields the total cost.

Listing 14: QAP model (raw API)

```
CPSolver cp = makeSolver();
CPIntVar[] x = makeIntVarArray(cp, n, n);

// build the objective function
CPIntVar[] weightedDist = new CPIntVar[n * n];
int ind = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        CPIntVar distXiXj = element(d, x[i], x[j]);
        weightedDist[ind] = mul(distXiXj, w[i][j]);
        ind++;
    }
}
CPIntVar totCost = sum(weightedDist);
Objective obj = cp.minimize(totCost);

DFSearch dfs = makeDfs(cp, firstFailBinary(x));
```

Storing and updating the best solution. Because LNS iteratively improves a solution, the user must maintain an external array `xBest` that records the best assignment found so far. The `onSolution` callback, shown in Listing 15, is invoked *every time* the search finds a feasible solution whose cost improves upon the current bound (since we are optimizing, the branch-and-bound mechanism guarantees this). Inside the callback, we copy the value of each variable (`x[i].min()` returns the unique value when the variable is fixed) into `xBest`. The array `xBest` is initialized to the identity permutation $\langle 0, 1, \ldots, n-1 \rangle$, which serves as the starting point for the first LNS iteration. Note that `xBest` lives *outside* the solver's reversible state: it is a plain Java array that persists across backtracks and restarts, so it always reflects the globally best solution encountered throughout the entire LNS process.

Listing 15: Best-solution tracking

```
int[] xBest = IntStream.range(0, n).toArray();

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

The LNS loop. Listing 16 shows the main LNS loop. At each restart, the method `optimizeSubjectTo` performs three steps:

1. The state manager *saves* the current solver state (all variable domains, propagation queues, etc.).

2. The `subjectTo` closure is executed: it posts temporary equality constraints that fix a randomly selected 95 % of the variables to their values in `xBest`. The remaining 5 % are left free—this is the *neighborhood* that the solver will explore.

3. A branch-and-bound search is launched within this restricted subspace, limited to `failureLimit` failures. Every improving solution found triggers the `onSolution` callback registered above, updating `xBest`.

After the search completes (or reaches the failure limit), the state manager *restores* the solver to the saved state, automatically retracting all temporary constraints. The next iteration therefore starts from a clean slate, with `xBest` reflecting the best solution found so far across all previous iterations.

Listing 16: LNS loop for the QAP

```
int nRestarts = 100;
int failureLimit = 100;
Random rand = new Random(0);

for (int i = 0; i < nRestarts; i++) {
    dfs.optimizeSubjectTo(obj,
        stats -> stats.numberOfFailures() >= failureLimit,
        () -> {
            // Fix 95% of the variables to their best-known value
            for (int j = 0; j < n; j++) {
                if (rand.nextInt(100) < 95) {
                    // after optimizeSubjectTo these constraints
                    // are automatically removed
                    cp.post(eq(x[j], xBest[j]));
                }
            }
        }
    );
}
```

If the `subjectTo` closure itself leads to an inconsistency (e.g., fixing two variables to the same value under an `allDifferent` constraint), the resulting `InconsistencyException` is silently caught and the iteration is simply skipped.

Table 1 illustrates a hypothetical first four LNS iterations on a small QAP instance with 12 facilities. At each iteration, variables marked with "$\star$" are relaxed while the others are fixed to their best-known value. The objective steadily decreases as the solver finds improving solutions in successive neighborhoods.

6 Scheduling with Conditional Time Intervals

Scheduling is a major application area for constraint programming as illustrated in [8].

Modern scheduling requires support for optional tasks (tasks that may or may not be executed), alternative resources (a task can be processed on one of several machines), and complex resource interactions (producers and consumers). MaxiCP supports all of these through conditional time-interval variables and cumulative function expressions, following the modeling paradigm introduced in [17] and implemented through the Generalized Cumulative constraint [5].

6.1 Conditional Time-Interval Variables

A conditional time-interval variable represents the execution of a task that may or may not occur. Its domain is a subset of $\{\perp\} \cup \{[s, e) \mid s \leq e,\ s, e \in \mathbb{Z}\}$, where $\perp$ denotes absence [16].

In MaxiCP, a `CPIntervalVar` has the following attributes:

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 1: Hypothetical first four LNS iterations on a QAP instance with 12 facilities. At each iteration the table shows the current best assignment and the relaxed neighborhood. Variables marked ⋆ are free; the others are fixed to their best-known value.

- $p \in \{true, false\}$: the presence status. If $p = false$, the task is absent ($\perp$).
- $s \in [\underline{s}, \overline{s}]$: the start time variable.
- $e \in [\underline{e}, \overline{e}]$: the end time variable.
- $d \in [\underline{d}, \overline{d}]$: the duration variable, satisfying $s + d = e$.

Bound consistency on the relation $s + d = e$ is maintained internally. The domain becomes $\perp$ (absent) whenever any attribute becomes empty, which triggers an inconsistency if the task is required to be present.

6.2 Creating Interval Variables

The `CPFactory` provides several factory methods for creating interval variables:

Listing 17: Creating interval variables (raw API)

```
CPSolver cp = makeSolver();

// Present task with fixed duration
CPIntervalVar task = makeIntervalVar(cp, false, duration);

// Optional task with flexible duration
CPIntervalVar opt = makeIntervalVar(cp);

// Present task with duration range [lengthMin, lengthMax]
CPIntervalVar t = makeIntervalVar(cp, false, lengthMin, lengthMax);

// Array of interval variables
CPIntervalVar[] tasks = makeIntervalVarArray(cp, n);
for (int i = 0; i < n; i++) {
    tasks[i].setLength(duration[i]);
    tasks[i].setPresent();
}
```

The second argument of `makeIntervalVar` is a boolean indicating whether the task is optional (`true`) or present (`false`). Methods `setLength` and `setPresent` can be called on the variable to fix the duration and mark the task as mandatory.

6.3 Modeling the Job-Shop

The classical Job-Shop Scheduling Problem (JSP) consists of n jobs, each comprising a sequence of operations that must be processed on a specific machine. Each operation has a fixed duration. Two types of constraints apply: (i) *precedence*: within each job, operations must be executed in order; (ii) *no-overlap*: no two operations on the same machine can overlap. The objective is to minimize the makespan. Listing 18 shows the MaxiCP model using the raw API.

Listing 18: Job-Shop model (raw API)

```
CPSolver cp = makeSolver();

CPIntervalVar[][] activities = new CPIntervalVar[nJobs][nMachines];
for (int j = 0; j < nJobs; j++) {
    for (int m = 0; m < nMachines; m++) {
        activities[j][m] = makeIntervalVar(cp, false, duration[j][m]);
    }
}

// Precedences within each job
for (int j = 0; j < nJobs; j++) {
    for (int m = 1; m < nMachines; m++) {
        cp.post(endBeforeStart(activities[j][m - 1], activities[j][m]));
    }
}

// No overlap on each machine
CPIntervalVar[][] toRank = new CPIntervalVar[nMachines][];
for (int m = 0; m < nMachines; m++) {
    ArrayList<CPIntervalVar> onMachine = new ArrayList<>();

    for (int j = 0; j < nJobs; j++) {
        for (int i = 0; i < nMachines; i++) {
            if (machine[j][i] == m) {
                onMachine.add(activities[j][i]);
            }
        }
    }

    toRank[m] = onMachine.toArray(new CPIntervalVar[0]);
    cp.post(noOverlap(toRank[m]));
}

CPIntVar makespan = makespan(
  Arrays.stream(activities)
        .map(job -> job[nMachines - 1])
        .toArray(CPIntervalVar[]::new)
);

Objective obj = cp.minimize(makespan);

DFSearch dfs = makeDfs(cp, new Rank(toRank));
dfs.optimize(obj);
```

Each `CPIntervalVar` represents an operation with a fixed duration. The `endBeforeStart(a, b)` constraint enforces that operation a ends before operation b starts. The `noOverlap` constraint ensures that activities sharing a machine do not overlap in time [2]. The `makespan` helper returns an integer variable equal to the latest end time among the given intervals. The `Rank` search heuristic selects the `noOverlap` group with the least slack and decides which activity goes next on that machine; this ranking-based strategy is well suited for job-shop problems where the key decision is the ordering of operations on each machine.

[2]This is enforced in MaxiCP filtering algorithms of [37]

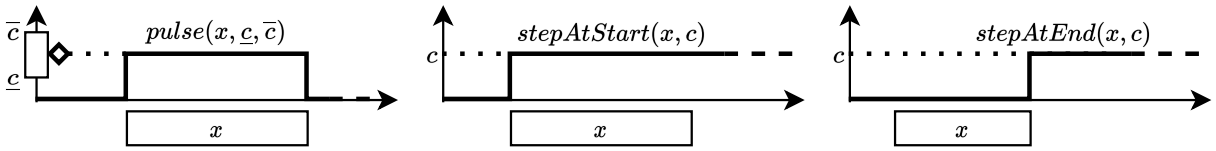

Figure 1: Elementary cumulative functions: `pulse`, `stepAtStart`, and `stepAtEnd`. Each function receives a conditional time-interval variable x_i and a height c_i. (Figure from [5].)

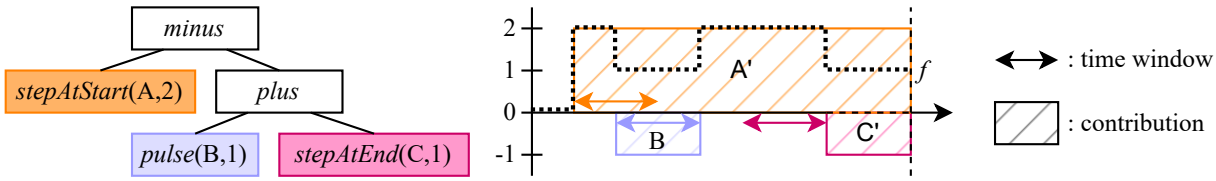

Figure 2: Left: AST for the cumulative function $f = stepAtStart(A, 2) - (pulse(B, 1) + stepAtEnd(C, 1))$. Right: the resulting cumulative function profile. Solid rectangles represent the time windows of the tasks; hatched areas show their contributions to the profile. (Figure from [5].)

6.4 Cumulative Function Expressions

For modeling cumulative resources, MaxiCP supports *cumulative function expressions* [17]. A cumulative function represents a time-varying resource profile, built compositionally from elementary functions illustrated in Figure 1:

- `pulse(x, h)`: contributes a height h during the execution of task x.
- `stepAtStart(x, h)`: adds a height h from the start of task x until the horizon.
- `stepAtEnd(x, h)`: adds a height h from the end of task x until the horizon.
- `flat()`: a zero function (identity for summation).

Cumulative functions can be combined using the following operators as illustrated in Figure 2:

- `sum(f1, f2, ...)`: sum of multiple cumulative functions.
- `minus(f1, f2)`: difference of two functions.

The resulting cumulative function can be viewed as a cumulative expression or AST that can then be flattened to extract individual cumulative tasks as illustrated in the left part of Figure 2 and explained in details in [5].

The height of a cumulative function can be bounded:

- `le(f, maxCapa)`: the function f must not exceed `maxCapa` at any time where at least one task executes.
- `alwaysIn(f, minCapa, maxCapa)`: the function f must stay between `minCapa` and `maxCapa`.

Internally, these constraints are compiled into a single `GeneralizedCumulative` constraint [5], which accepts tasks with variable, possibly negative heights. The task variables are extracted from the cumulative function expression by traversing its AST through a flattening process.

It is important to note that the `GeneralizedCumulative` constraint is more general than the classical cumulative constraint, which only allows positive heights and fixed tasks. This constraint enforces the cumulative bounds at every time point where at least one task executes. The formal semantics is given by the following definition.

Definition 1 (`GeneralizedCumulative`)**.** *The* `GeneralizedCumulative`$(T, \underline{C}, \overline{C})$ *constraint enforces that the cumulated heights of the tasks in* T *lie within* $[\underline{C}, \overline{C}]$ *at every time point at which at least one task executes:*

$$\sum_{i \in R | s_i \leq \tau < e_i} c_i \in [\underline{C}, \overline{C}] \qquad \forall \tau \in \bigcup_{i \in R} [s_i, e_i)$$

where R *is the set of required (present) tasks and* c_i *is the height of task* i*.*

6.5 Modeling the RCPSP

The Resource-Constrained Project Scheduling Problem (RCPSP) is a classical scheduling benchmark. Each activity has a fixed duration, consumes one or more resources, and must respect precedence constraints. Unlike the Job-Shop where machines impose a *no-overlap* requirement (unit capacity), the RCPSP involves *cumulative* resources with capacity greater than one: multiple activities may execute simultaneously on the same resource, as long as their total consumption does not exceed the capacity. This is modeled using cumulative function expressions. Listing 19 shows the MaxiCP model using the raw API.

Listing 19: RCPSP model (raw API)

```
CPSolver cp = makeSolver();

CPIntervalVar[] tasks = makeIntervalVarArray(cp, nActivities);
for (int i = 0; i < nActivities; i++) {
    tasks[i].setLength(duration[i]);
    tasks[i].setPresent();
}

// Cumulative resources
CPCumulFunction[] resources = new CPCumulFunction[nResources];
for (int r = 0; r < nResources; r++) {
    resources[r] = new CPFlatCumulFunction();
}

for (int i = 0; i < nActivities; i++) {
    for (int r = 0; r < nResources; r++) {
        if (consumption[r][i] > 0) {
            resources[r] = new CPPlusCumulFunction(
                resources[r],
                new CPPulseCumulFunction(tasks[i], consumption[r][i])
            );
        }
    }
}

for (int r = 0; r < nResources; r++) {
    cp.post(le(resources[r], capa[r]));
}

// Precedences
for (int i = 0; i < nActivities; i++) {
    for (int k : successors[i]) {
        cp.post(endBeforeStart(tasks[i], tasks[k]));
    }
}

CPIntVar makespan = makespan(tasks);
Objective obj = cp.minimize(makespan);

DFSearch dfs = makeDfs(cp, fds(tasks));
dfs.optimize(obj);
```

The resource profile is built compositionally: `CPFlatCumulFunction` represents the zero function, and for each activity consuming resource r, a `CPPulseCumulFunction` (contributing the consumption height while the task executes) is added using `CPPlusCumulFunction`. The `le(resource, capa)` constraint ensures the cumulative profile never exceeds the capacity. The search uses the Failure-Directed Search heuristic `fds` [19].

6.6 Producer-Consumer Scheduling

Cumulative functions excel at modeling producer-consumer problems. Consider a reservoir with limited capacity: producer tasks add to the level at their completion, and consumer tasks remove from it at their start. Listing 20 shows this using the raw API.

Listing 20: Producer-consumer with cumulative functions (raw API)

```
CPCumulFunction reservoir = new CPFlatCumulFunction();

for (int i = 0; i < nProducers; i++) {
    reservoir = new CPPlusCumulFunction(
        reservoir,
        stepAtEnd(producerTask[i], producerQty[i])
    );
}

for (int i = 0; i < nConsumers; i++) {
    reservoir = new CPPlusCumulFunction(
        reservoir,
        stepAtStart(consumerTask[i], -consumerQty[i])
    );
}

cp.post(alwaysIn(reservoir, 0, capacity));
```

The `stepAtEnd(task, qty)` adds `qty` to the function at the task's end (the producer finishes and adds inventory). The `stepAtStart(task, -qty)` subtracts at the task's start (the consumer begins and removes inventory). The `alwaysIn` constraint ensures the reservoir level stays between 0 and the capacity at all times.

7 Sequence Variables for Routing

Vehicle Routing Problems (VRP) are among the most important applications of constraint programming. Traditional CP models for routing use *successor arrays*, where an integer variable $succ_i$ represents the direct successor of node i in the tour. While functional, this representation has significant limitations: it does not naturally support optional visits, and it cannot support insertion-based search heuristics, which are critical for quickly finding good solutions in practice.

MaxiCP introduces *sequence variables* [6], a dedicated computational domain for routing and sequencing problems. Unlike the successor model, sequence variables represent a partial, growing path through node insertions, and natively support optional visits and insertion-based branching.

7.1 Domain of a Sequence Variable

A sequence variable $\mathbf{S}$ represents an ordered sequence of pairwise distinct nodes from a set $\mathcal{V}$, starting at an origin α and ending at a destination ω. Its domain is represented intensionally as a tuple $\mathcal{D} = \langle R, X, \mathbf{s}, N \rangle$, where:

- $R \subseteq \mathcal{V}$ is the set of *required* nodes that must appear in every feasible sequence.
- $X \subseteq \mathcal{V}$ is the set of *excluded* nodes that may not appear in any feasible sequence.
- $\mathbf{s}$ is a mandatory *partial sequence*: a subsequence that must appear in every feasible sequence.
- $N \subseteq \mathcal{V}^3$ is a set of *NotBetween* triples: for each $(i, j, k) \in N$, node j is forbidden from appearing between i and k.

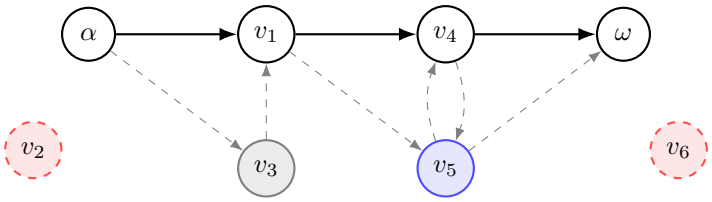

Figure 3: Domain of a sequence variable. Solid arrows form the partial sequence $(\alpha, v_1, v_4, \omega)$. Dashed arrows indicate feasible insertions. Node v_3 is possible (gray), v_5 is required (blue border), and v_2, v_6 are excluded (red, dashed border).

Definition 2 (Sequence Variable Domain)**.** *The set of values represented by* $\langle R, X, \mathbf{s}, N\rangle$ *is:*

$$[\![\langle R, X, \mathbf{s}, N\rangle]\!] = \left\{ \mathbf{t} \in \mathcal{S}(\mathcal{V}) \;\middle|\; \begin{array}{l} \mathbf{t} \textit{ is a simple } (\alpha,\omega)\textit{-path}, \\ R \subseteq \text{nodes}(\mathbf{t}),\ \text{nodes}(\mathbf{t}) \cap X = \varnothing, \\ \mathbf{s} \preceq \mathbf{t}, \\ \forall (i,j,k) \in N:\ \neg(i \prec j \prec k \textit{ in } \mathbf{t}) \end{array} \right\}$$

The four elementary domain restrictions are monotonic: adding elements to R, X, or N, or extending $\mathbf{s}$, yields a smaller (more constrained) domain. This provides a natural basis for incremental propagation.

Nodes are classified into four categories, illustrated in Figure 3:

- **Member**: nodes currently in the partial sequence $\mathbf{s}$.
- **Possible**: nodes that may or may not be in the final sequence (neither required nor excluded, and not yet inserted).
- **Required**: nodes that must be in the final sequence but are not yet inserted into $\mathbf{s}$.
- **Excluded**: nodes that have been removed from all feasible sequences.

7.2 Compact Encoding

The domain $\mathcal{D}$ is stored as a graph with $O(|\mathcal{V}|^2)$ memory. The partial sequence $\mathbf{s}$ is maintained as a doubly-linked list. For each node v not in $\mathbf{s}$, the set of feasible insertion positions is represented as the set of member nodes after which v can be inserted. A pair (u, v) means "v can be inserted immediately after u in $\mathbf{s}$." The NotBetween triples N are encoded by removing the corresponding insertion pair, under the restriction that the extremities of each NotBetween belong to $\mathbf{s}$.

Domain updates are:

- `insert(pred, node)`: insert `node` after `pred` in $\mathbf{s}$.
- `exclude(node)`: remove `node` from all feasible sequences.
- `require(node)`: mark `node` as required.
- `removeInsert(pred, node)`: remove the insertion pair (`pred`, `node`).

All updates are reversible through the state management system.

7.3 Global Constraints for Sequence Variables

MaxiCP provides several global constraints tailored for sequence variables:

`distance(SeqVar seqVar, int[][] dist, IntExpression totalDist).` It links the total distance of the route to the total distance variable, based on a distance matrix.

`transitionTimes(SeqVar seqVar, IntExpression[] time, int[][] dist).` It enforces temporal consistency along the sequence `seqVar` using transition times between nodes. Each node v is associated with a variable $time[v]$ representing the start time of its service (typically constrained by a time window). For every pair of nodes (u, v) such that u precedes v in `seqVar`, the constraint enforces:

$$time[u] + d_{u,v} \leq time[v].$$

This ensures that node v cannot be started before reaching it from u. The use of an inequality allows waiting at node v (i.e., arriving earlier than its earliest start time). Nodes that are not present in `seqVar` are not constrained.

`cumulative(SeqVar seqVar, int[] starts, int[] ends, int[] load, int capacity).` It enforces capacity constraints for pickup-and-delivery activities along the sequence `seqVar`. Introduced in [38], it allows the modeling of the evolution of the vehicle load along the route: picking up a request increases the load, and delivering it decreases the load, while ensuring feasibility at all times.

Each activity i is defined by a pickup node $starts[i]$, a delivery node $ends[i]$, and a load $load[i]$ that is carried from the pickup to the delivery. The constraint ensures that:

- Capacity constraint. At every node v in `seqVar`, the total load of all active activities does not exceed the vehicle capacity:

$$\sum_{i | starts[i] \preceq v \prec ends[i]} load[i] \leq capacity.$$

  An activity contributes to the load between its pickup and its delivery.

- Pairing of nodes. For each activity i, the pickup and delivery nodes are either both present in `seqVar` or both absent.

- Precedence. If an activity is performed, its pickup node must appear before its delivery node in `seqVar`.

7.4 Modeling the TSPTW

The Traveling Salesman Problem with Time Windows (TSPTW) is a classical routing problem. In MaxiCP, it can be modeled using either integer variables or sequence variables. We first show the integer-variable model (Listing 21), which uses a position-based formulation, and then the sequence-variable model (Listing 22) used in [15].

Listing 21: TSPTW with integer variables (raw API)

```
CPSolver cp = makeSolver();

// x[i] is the node visited at position i
CPIntVar[] x = makeIntVarArray(cp, n, n);
// arrival[i] is the arrival time at position i
CPIntVar[] arrival = makeIntVarArray(cp, n, horizon);

cp.post(allDifferent(x));
cp.post(eq(x[0], 0));          // start at depot
cp.post(eq(x[n-1], n-1));      // end at depot copy
cp.post(eq(arrival[0], 0));    // depart at time 0

for (int i = 0; i < n; i++) {
    CPIntVar earliest = element(inst.earliest, x[i]);
    CPIntVar latest = element(inst.latest, x[i]);
    cp.post(le(earliest, arrival[i]));
    cp.post(le(arrival[i], latest));
}

CPIntVar[] transition = new CPIntVar[n - 1];
for (int i = 0; i < n - 1; i++) {
    transition[i] = element(distMatrix, x[i], x[i+1]);
    cp.post(le(sum(arrival[i], transition[i]),
               arrival[i+1]));
}

CPIntVar distance = sum(transition);
DFSearch search = makeDfs(cp, staticOrderBinary(x));
search.optimize(cp.minimize(distance));
```

Here, `element(T, x[i])` is the *element* constraint that returns $T[x_i]$, enabling indexing into data arrays with decision variables. The two-dimensional variant `element(T, x[i], x[i+1])` retrieves $T[x_i][x_{i+1}]$.

The TSPTW can also be modeled with a sequence variable, which provides a more natural representation. Listing 22 shows this model.

Listing 22: TSPTW with a sequence variable (raw API)

```
CPSolver cp = makeSolver();

CPSeqVar tour = makeSeqVar(cp, n, 0, n - 1);
for (int i = 0; i < n; i++)
    tour.require(i);  // all nodes must be visited

CPIntVar[] time = makeIntVarArray(cp, n, horizon);
cp.post(eq(time[0], 0));
for (int i = 1; i < n; i++) {
    time[i].removeAbove(latest[i]);
    time[i].removeBelow(earliest[i]);
}

cp.post(new TransitionTimes(tour, time, distMatrix));

CPIntVar totDist = makeIntVar(cp, 0, 100000);
cp.post(new Distance(tour, distMatrix, totDist));

DFSearch dfs = makeDfs(cp, /* insertion search */);

dfs.optimize(cp.minimize(totDist));
```

The `CPSeqVar` represents a path from node 0 (depot) to node $n-1$ (depot copy). Each node is required using `tour.require(i)`. The `TransitionTimes` constraint links the time-window variables and enforces travel times between consecutive nodes. The `Distance` constraint bounds the total distance.

7.5 Custom Insertion-Based Search

Built-in heuristics such as `firstFailBinary(seqVars)` (Section 5) are convenient, but routing problems often benefit from a custom insertion-based search. Writing one requires querying the sequence variable's domain at each search node. The key API methods are:

- `tour.isFixed()`: returns `true` when all nodes are either members or excluded.
- `tour.fillNode(nodes, INSERTABLE)`: fills the array with nodes that are not yet in the partial sequence and returns their count.
- `tour.nInsert(node)`: returns the number of feasible insertion points for a node.
- `tour.fillInsert(node, preds)`: fills the array with the member nodes after which `node` can be inserted, and returns their count.
- `tour.memberAfter(pred)`: returns the current successor of `pred` in the partial sequence.

Listing 23 shows a custom branching for the TSP. At each search node, the heuristic (i) selects the insertable node with the fewest insertion points (first-fail on insertions), (ii) among its feasible insertion positions, picks the one with the smallest *detour cost* $d_{pred,node} + d_{node,succ} - d_{pred,succ}$, and (iii) creates a binary branching: insert the node at that position on the left, or forbid that placement with a `notBetween` constraint on the right.

Listing 23: Custom insertion-based search for the TSP (raw API)

```
int[] nodes = new int[n];
DFSearch dfs = makeDfs(cp, () -> {
    if (tour.isFixed()) return EMPTY;

    // Variable selection: node with fewest insertion points
    int nInsertable = tour.fillNode(nodes, INSERTABLE);
    int node = selectMin(nodes, nInsertable,
        i -> true, tour::nInsert).getAsInt();

    // Value selection: insertion with smallest detour
    int nInsert = tour.fillInsert(node, nodes);
    int bestPred = selectMin(nodes, nInsert,
        pred -> true,
        pred -> {
            int succ = tour.memberAfter(pred);
            return dist[pred][node] + dist[node][succ]
                 - dist[pred][succ];
        }).getAsInt();
    int succ = tour.memberAfter(bestPred);

    // Binary branch: insert or forbid this placement
    return branch(
        () -> cp.post(insert(tour, bestPred, node)),
        () -> cp.post(notBetween(tour, bestPred, node, succ))
    );
});
```

The `insert(tour, pred, node)` constraint inserts `node` immediately after `pred` in the partial sequence, so that the path becomes $\ldots \to \mathit{pred} \to \mathit{node} \to \mathit{succ} \to \ldots$. The `notBetween(tour, pred, node, succ)` constraint forbids `node` from appearing between `pred` and `succ` in any feasible sequence, effectively pruning that insertion point without committing to a specific alternative. This binary branching mirrors the assign/remove pattern of `heuristicBinary` for integer variables: the left branch commits to the best insertion, while the right branch eliminates it and lets the search explore other placements.

7.6 Vehicle Routing with Sequence Variables

For multi-vehicle routing, sequence variables truly shine. Listing 24 shows a Capacitated Vehicle Routing Problem with Time Windows (CVRPTW) model using the raw API, based on the Solomon benchmarks.

Listing 24: CVRPTW with sequence variables (raw API)

```
CPSolver cp = makeSolver();
CPSeqVar[] vehicles = new CPSeqVar[nVehicle];
CPIntVar[] time = new CPIntVar[nNode];
CPIntVar[] distance = new CPIntVar[nVehicle];

for (int v = 0; v < nVehicle; v++) {
    vehicles[v] = makeSeqVar(cp, nNode,
        nRequest + v * 2, nRequest + v * 2 + 1);
    distance[v] = makeIntVar(cp, 0, depotEnd);
}
for (int i = 0; i < nRequest; i++)
    time[i] = makeIntVar(cp, twStart[i], twEnd[i]);
for (int i = nRequest; i < nNode; i++)
    time[i] = makeIntVar(cp, depotStart, depotEnd);

for (int v = 0; v < nVehicle; v++) {
    cp.post(new TransitionTimes(
        vehicles[v], time, distMatrix, duration));
    // Capacity + pickup-before-delivery + same vehicle
    cp.post(new Cumulative(
        vehicles[v], pickups, drops, load, capacity));
    cp.post(new Distance(
        vehicles[v], distMatrix, distance[v]));
}
CPIntVar sumDist = sum(distance);

// Each node visited exactly once across all vehicles
for (int node = 0; node < nNode; node++) {
    CPIntVar[] visits = new CPIntVar[nVehicle];
    for (int v = 0; v < nVehicle; v++)
        visits[v] = vehicles[v].getNodeVar(node)
                               .isRequired();
    cp.post(new Sum(visits, 1));
}

DFSearch search = makeDfs(cp, /* insertion search */);
search.optimize(cp.minimize(sumDist));
```

Each vehicle is represented by a `CPSeqVar` with its own start and end depot nodes. The `getNodeVar(node).isRequired()` method returns a 0/1 `CPIntVar` indicating whether the node is required (assigned) to that vehicle. The capacity is enforced by summing the loads of the required nodes. The constraint $\sum_v \texttt{visits}_v = 1$ ensures each customer is served by exactly one vehicle.

7.7 Pickup and Delivery with Sequence Variables

The Dial-A-Ride Problem (DARP) is a Pickup and Delivery Problem with time windows, load capacity, and ride-time constraints. Each request has a pickup node and a delivery node; the vehicle must visit the pickup before the delivery while respecting its capacity. Listing 25 shows the model using the raw API.

Listing 25: DARP with sequence variables and cumulative load (raw API)

```
CPSolver cp = makeSolver();
CPSeqVar[] routes = new CPSeqVar[nVehicle];
CPIntVar[] time = new CPIntVar[n];
CPIntVar[] distance = new CPIntVar[nVehicle];

for (int v = 0; v < nVehicle; v++) {
    routes[v] = makeSeqVar(cp, n,
        rangeDepot + v, rangeDepot + nVehicle + v);
    distance[v] = makeIntVar(cp, 0, maxDist);
}
for (int i = 0; i < n; i++)
    time[i] = makeIntVar(cp, twStart[i], twEnd[i]);

int[] pickups = IntStream.range(0, nRequest).toArray();
int[] drops = IntStream.range(nRequest, 2*nRequest).toArray();

for (int v = 0; v < nVehicle; v++) {
    cp.post(new TransitionTimes(
        routes[v], time, distMatrix, duration));
    // Capacity + pickup-before-delivery + same vehicle
    cp.post(new Cumulative(
        routes[v], pickups, drops, load, capacity));
    cp.post(new Distance(
        routes[v], distMatrix, distance[v]));
}

// Max ride time: time[drop] <= time[pickup] + dur + maxRide
for (int i = 0; i < nRequest; i++)
    cp.post(new LessOrEqual(time[i + nRequest],
        plus(time[i], duration[i] + maxRideTime)));

// Each node visited exactly once
for (int node = 0; node < n; node++) {
    CPIntVar[] visits = new CPIntVar[nVehicle];
    for (int v = 0; v < nVehicle; v++)
        visits[v] = routes[v].getNodeVar(node).isRequired();
    cp.post(new Sum(visits, 1));
}

CPIntVar sumDist = sum(distance);
DFSearch search = makeDfs(cp, /* insertion search */);
search.optimize(cp.minimize(sumDist));
```

The `Cumulative` constraint on a sequence variable is particularly powerful: it simultaneously enforces the vehicle capacity, ensures each request's pickup precedes its delivery, and guarantees that both nodes of a request are assigned to the same vehicle. The `LessOrEqual` constraint enforces the maximum ride time for each request.

7.8 LNS with Sequence Variables

Sequence variables are ideally suited for Large Neighborhood Search. A destroy-and-repair cycle consists of:

1. **Destroy**: select a set of nodes to relax.

2. **Repair**: enforce the previous solution except for relaxed nodes, then search for new insertions.

The key primitive is `RelaxedSequence`: given the current best tour stored in an array `bestTour` and a set of relaxed nodes, it constrains the sequence variable to follow the previous ordering for all non-relaxed nodes, while leaving the relaxed nodes free for re-insertion. Listing 26 shows the LNS loop using the raw API.

Listing 26: LNS with sequence variables (raw API)

```
int[] bestTour = IntStream.range(0, n + 1).toArray();

dfs.onSolution(() -> {
    tour.fillNode(bestTour, SeqStatus.MEMBER_ORDERED);
});

Random random = new Random(42);
for (int iter = 0; iter < 100; iter++) {
    dfs.optimizeSubjectTo(obj, s -> false, () -> {
        Set<Integer> relaxed = randomSubset(random, 0, n, 5);
        cp.post(new RelaxedSequence(tour, bestTour, relaxed));
    });
}
```

At each iteration, `optimizeSubjectTo` saves the solver state, posts the `RelaxedSequence` constraint that fixes all non-relaxed nodes in their best-known order, searches for improving solutions, and then restores the state. This follows the same pattern as the QAP LNS loop (Section 5.4), but the `RelaxedSequence` constraint is specifically designed for sequence variables.

8 Symbolic Modeling with MaxiCP-Modelling

Having covered the raw implementation layer—state management, propagation, search, sequence variables, scheduling, and multi-objective support—we now turn to the symbolic modeling layer. As introduced in Section 2.1, MaxiCP provides a higher-level API that separates the *definition* of a model from its *resolution*. This is achieved through a symbolic modeling layer called MaxiCP-Modelling [7], which treats models as first-class, immutable data structures.

8.1 Models as Functional Linked Lists

In MaxiCP-Modelling, a `SymbolicModel` is an immutable record consisting of a constraint, a pointer to a parent model, and a reference to the `ModelProxy`. Adding a constraint to a model returns a *new* model node, leaving the original unchanged:

Listing 27: The SymbolicModel record

```
public record SymbolicModel(
    Constraint constraint,
    SymbolicModel parent,
    ModelProxy modelProxy
) implements Model, Iterable<Constraint> {
    public SymbolicModel add(Constraint c) {
        return new SymbolicModel(c, this, modelProxy);
    }
}
```

Because models are immutable linked lists, adding a constraint is an $O(1)$ operation. The complete list of constraints can be recovered by traversing the chain to the root. This design enables the creation of *model trees*, where each branch represents a different sub-problem, reformulation, or neighborhood.

8.2 The `ModelDispatcher`

The `ModelDispatcher` is the primary user-facing class for the symbolic layer. It maintains a thread-local reference to the current model and acts as both a variable factory and a model proxy. When the user calls `model.add(constraint)`, the dispatcher appends the constraint to the current symbolic model:

Listing 28: Creating a model with the ModelDispatcher

```
ModelDispatcher model = Factory.makeModelDispatcher();
IntVar[] x = model.intVarArray(n, n);
model.add(allDifferent(x));
model.add(eq(x[0], 0));
```

Variables are symbolic objects that do not belong to any concrete solver until concretization. This allows the same variables to be shared across multiple concrete solver instances.

8.3 Concretization

To solve a model, it must be *concretized* into a solver engine. The `runCP` method on `ModelDispatcher` handles this:

Listing 29: Concretizing and solving a model

```
model.runCP(cp -> {
    DFSearch search = cp.dfSearch(firstFailBinary(x));
    SearchStatistics stats = search.solve();
});
```

Internally, this creates a `ConcreteCPModel` that:

1. Creates a fresh `MaxiCP` solver instance with a `Trailer`.
2. Iterates over the symbolic constraint list and *instantiates* each one in the underlying CP engine. Symbolic expressions (e.g., `Sum`, `Element1D`) are translated into concrete CP variables and constraints through pattern matching in the `ConcreteCPModel` class.

3. Runs the initial fixed-point propagation.

The mapping between symbolic and concrete variables is maintained in reversible state maps, enabling incremental constraint addition during search.

8.4 Model Transformations and Reformulations

Because models are immutable trees, they support powerful compositional operations:

Neighborhoods for LNS. A neighborhood is simply a model extension. Given a base model and a solution, one creates a new model by adding fixing constraints:

Listing 30: LNS with symbolic models

```
Model relaxed = base;
for (IntVar x : variables) {
    if (rand.nextDouble() > 0.1)
        relaxed = relaxed.add(eq(x, solution.get(x)));
}
relaxed.runCP(cp -> ...); // solve the neighborhood
```

Each neighborhood is a different branch of the model tree. The base model is never modified. This approach to LNS complements the raw-level `optimizeSubjectTo` technique presented in Section 5.4: while the raw API relies on save/restore of the solver state, the symbolic approach creates entirely independent model branches that can be solved concurrently.

Hybridization. Although not implemented in MaxiCP the same symbolic model could be transformed into input for a MIP solver (through linearization of supported constraints) or solved concurrently by both a CP and a MIP engine. We refer to [3] for a detailed case study of CP/MIP hybridization using symbolic modeling.

8.5 Embarrassingly Parallel Search

Embarrassingly Parallel Search (EPS) [39, 40] is a simple yet effective approach to parallelizing constraint solving. The idea is to decompose the original problem into a large number of independent sub-problems that can be solved concurrently with no communication between workers. Concretely, the search tree is explored up to a fixed depth d using the normal branching heuristic. Each leaf of this depth-limited exploration defines a sub-problem: the original model augmented with the branching decisions made along the path from the root to that leaf. These sub-problems are then dispatched to a thread pool for independent resolution. Because the sub-problems are disjoint (they partition the search space), the total solution count—or the global optimum—can be computed by simply aggregating the individual results.

The symbolic modeling layer of MaxiCP makes EPS particularly natural to implement. Since models are immutable linked lists, the branching decisions applied during the decomposition phase produce new symbolic model nodes without modifying the original model. At each leaf of the depth-limited search, a call to `cp.symbolicCopy()` captures the current sub-problem as a standalone symbolic model. This model can then be concretized and solved in a separate thread, with no shared mutable state.

Listing 31 illustrates EPS on the N-Queens problem. The model and branching heuristic are identical to those shown in Listing 2. The decomposition wraps the branching in a `LimitedDepthBranching` combinator that stops at depth 10, causing the search to treat each leaf as a "solution." For each such leaf, `cp.symbolicCopy()` extracts the corresponding sub-problem,

which is submitted to a fixed thread pool of 8 workers. Each worker concretizes the sub-problem into a fresh solver instance and runs a complete search. Finally, the solution counts are summed.

Listing 31: N-Queens solved with EPS

```
// model and branching defined as in Listing 2
ExecutorService executor = Executors.newFixedThreadPool(8);

Function<Model, SearchStatistics> epsSolve = (m) -> {
    return model.runAsConcrete(
        CPModelInstantiator.withTrailing, m, (cp) -> {
        DFSearch search = cp.dfSearch(branching);
        return search.solve();
    });
};

LinkedList<Future<SearchStatistics>> results = new LinkedList<>();
model.runCP((cp) -> {
    DFSearch search = cp.dfSearch(
        new LimitedDepthBranching(branching, 10));
    search.onSolution(() -> {
        Model m = cp.symbolicCopy(); // capture sub-problem
        results.add(executor.submit(
            () -> epsSolve.apply(m)));
    });
    search.solve(); // decompose
});

int total = 0;
for (var f : results)
    total += f.get().numberOfSolutions();
System.out.println("Total solutions: " + total);
executor.shutdown();
```

The key enabler is that `symbolicCopy()` returns a lightweight, immutable snapshot of the model at the current search node. Because each sub-problem is concretized independently, there is no shared solver state and therefore no synchronization overhead. The depth parameter d controls the granularity of the decomposition: a larger d produces more and smaller sub-problems, which improves load balancing at the cost of a longer decomposition phase.

Since symbolic models are immutable and variables are symbolic, the same model can also be safely shared across multiple threads for portfolio-style parallel search. Each thread independently concretizes the model and explores the search space with a different heuristic or random seed:

Listing 32: Portfolio parallel search with symbolic models

```
ModelDispatcher model = ...; // build the model
IntStream.range(0, nThreads).parallel().forEach(t -> {
    model.runCP(cp -> {
        DFSearch search = cp.dfSearch(
            randomizedFirstFail(x, new Random(t)));
        search.optimize(minimize(cost));
    });
});
```

8.6 Mixing Both Layers

Inside the `runCP` callback, the user has access to both the concrete model and the symbolic variables. The concrete model allows accessing the underlying `CPVar` objects, which is useful for implementing custom search heuristics that need direct access to domain internals.

9 Conclusion

We have presented MaxiCP, a modern, open-source constraint programming solver that bridges the gap between educational simplicity and industrial-grade performance. By building on the architectural clarity of MiniCP and incorporating ideas from OscaR and Objective-CP, MaxiCP introduces three key functionalities that set it apart for an open-source CP solver:

1. **Conditional time intervals and cumulative functions**: Full support for optional tasks, alternative resources, and producer-consumer scheduling through the Generalized Cumulative constraint.

2. **Sequence variables**: A dedicated computational domain for routing that supports optional visits, insertion-based heuristics, and LNS, with global constraints for distance, transition times, and cumulative load.

3. **Symbolic modeling**: Models as immutable functional linked lists enable model transformations, LNS neighborhoods, hybridization with MIP solvers, and embarrassingly parallel search.

The two-layer API design (raw and symbolic) makes MaxiCP accessible to both beginners learning CP and experts deploying it in production. The entire codebase is available as open-source under the MIT license.

Future work includes expanding the reformulation library (e.g., automatic linearization for MIP hybridization), integrating advanced black-box heuristics, and developing a MiniZinc backend.

References

[1] Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. doi: 10.1007/s12532-020-00190-7. URL `https://doi.org/10.1007/s12532-020-00190-7`.

[2] OscaR Team. OscaR: Scala in or. `https://bitbucket.org/oscarlib/oscar`, 2012.

[3] Pascal Van Hentenryck and Laurent Michel. The objective-cp optimization system. In *International Conference on Principles and Practice of Constraint Programming*, pages 8–29. Springer, 2013.

[4] MaxiCP Team. MaxiCP: A Not So Mini Constraint Programming Solver. `https://github.com/aia-uclouvain/maxicp`, 2024. Accessed: 2026-04-15.

[5] Pierre Schaus, Charles Thomas, and Roger Kameugne. Implementing cumulative functions with generalized cumulative constraints. *arXiv preprint arXiv:2410.02456*, 2024. URL `https://arxiv.org/abs/2410.02456`.

[6] Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. Sequence variables: A constraint programming computational domain for routing and sequencing. *Journal of Artificial Intelligence Research*, 2026. URL `https://arxiv.org/abs/2203.11666`.

[7] Guillaume Derval and Damien Ernst. Symbolism for modelling, reformulations, and parallelism: Maxicp-modelling. In *Proceedings of the 22nd Workshop on Constraint Modelling and Reformulation (ModRef 2023)*, 2023. URL `https://arxiv.org/abs/2308.06013`.

[8] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling: 20+ years of scheduling with constraints at ibm/ilog. *Constraints*, 23:210–250, 2018.

[9] Philip Kilby and Paul Shaw. An optimal algorithm for maximum cardinality 2-satisfiability. In *Principles and Practice of Constraint Programming–CP 2006*, pages 623–637. Springer, 2006.

[10] OptalCP. OptalCP: A Constraint Programming Scheduling Engine. `https://optalcp.com/`, 2024. Accessed: 2026-04-15.

[11] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver. *Journal of Open Source Software*, 7(78):4708, 2022.

[12] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www.gecode.dev*, pages 11–13, 2006.

[13] Google. OR-Tools: Google Operations Research Tools. `https://developers.google.com/optimization`, 2024. Accessed: 2026-04-15.

[14] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[15] Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. Sequence variables for routing problems. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022.

[16] Philippe Laborie and Jérôme Rogerie. Reasoning with conditional time-intervals. In *Proceedings of the 21st International FLAIRS Conference*, pages 555–560, 2008.

[17] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part ii: An algebraical model for resources. pages 555–560, 2009.

[18] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *International conference on principles and practice of constraint programming*, pages 140–148. Springer, 2015.

[19] Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 437–453. Springer, 2015.

[20] Christophe Lecoutre. ACE: A fast multithreaded constraint solver. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, 2023. `https://github.com/xcsp3team/ace`.

[21] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons, 2013.

[22] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

[23] Pierre Schaus. Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 971–978. IEEE Computer Society, 2013.

[24] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.

[25] Paul Shaw. A constraint for bin packing. In *International conference on principles and practice of constraint programming*, pages 648–662. Springer, 2004.

[26] Pierre Schaus et al. *Solving balancing and bin-packing problems with constraint programming.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2009.

[27] Guillaume Derval, Jean-Charles Régin, and Pierre Schaus. Improved filtering for the bin-packing with cardinality constraint. *Constraints*, 23(3):251–271, 2018.

[28] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 209–215, 1996.

[29] Margaux Schmied and Jean-Charles Régin. Efficient implementation of the global cardinality constraint with costs. *arXiv preprint arXiv:2502.02688*, 2025.

[30] Willem-Jan Van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4):347–373, 2006.

[31] Pierre Schaus, Pascal Van Hentenryck, and Alessandro Zanarini. Revisiting the soft global cardinality constraint. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 307–312. Springer, 2010.

[32] Alessandro Zanarini, Michela Milano, and Gilles Pesant. Improved algorithm for the soft global cardinality constraint. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 288–299. Springer, 2006.

[33] Jean-Guillaume Fages and Charles Prud'homme. Making the first solution good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1073–1077. IEEE, 2017.

[34] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[35] C. Le Pape, P. Couronne, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*, 1994.

[36] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 607–613, 1995.

[37] Petr Vilím. Global constraints in scheduling (phd thesis). 2007.

[38] Charles Thomas and Pierre Schaus. Insertion sequence variables for hybrid routing and scheduling problems. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 457–474, 2020.

[39] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *International Conference on Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[40] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly parallel search in constraint programming. *Journal of Artificial Intelligence Research*, 57:421–464, 2016.