Constraint-Based Local Search

9

Laurent Michel and Pascal Van Hentenryck

Contents

L. Michel (✉)
University of Connecticut, Storrs, CT, USA
e-mail: ldm@engr.uconn.edu

P. Van Hentenryck
University of Michigan, Ann Arbor, MI, USA
e-mail: pvanhent@umich.edu

R. Martí et al. (eds.), *Handbook of Heuristics*,
https://doi.org/10.1007/978-3-319-07124-4_7

Abstract

Constraint-Based Local Search emerged in the last decade as a framework for declaratively expressing hard combinatorial optimization problems and solve them with local search techniques. It delivers tools to practitioners that enables them to quickly experiment with multiple models, heuristics, and meta-heuristics, focusing on their application rather than the delicate minutiae of producing a competitive implementation. At its heart, the declarative models are reminiscent of the modeling facilities familiar to constraint programming, while the underlying computational model heavily depends on incrementality. The net result is a platform capable of delivering competitive local search solutions at a fraction of the efforts needed with a conventional approach delivering model-and-run to local search users.

Keywords

Constraint · Local search · Neighborhood · Synthetic search satisfaction · Optimization · Incremental model · Declarative

Introduction

Complete techniques such as Integer Programming and Constraint Programming typically offer optimality guarantees on the results they deliver but do not always scale to large instances. This explains the appeal of local search methods that deliver a different trade-off in the algorithmic design space, favoring scalability at the expense of guarantees. Local search algorithms apply to diverse application domains such as routing, scheduling, resource allocation, or rostering to name just a few. In most cases, local search techniques scale to truly large problem instances that are often out of scope for complete techniques and are capable of producing streams of solutions of improving quality.

Yet, while modeling and high-level tools are ubiquitous for Mixed Integer Programming and Constraint Programming, they have been relatively unexplored for local search until recently. This is primarily due to the fact that the separation of models and algorithms is not as simple in local search compared to MIP and CP which are declarative in nature. Indeed, the vast majority of papers discussing local search describe their solution in *algorithmic* terms rather than *declarative* terms like models, decision variables, and constraints. The 1990s witnessed a shift in interest and the emergence of multiple tools expressly dedicated to local search techniques. GSAT [13, 14] and WalkSAT [15] offered the initial impetus behind formulating problems with a simple language (in clausal form) and using generic local search algorithms operating on that encoding. Integer Optimization by Local Search [25] generalized this line of work to the richer language of integer programming.

Localizer [8,9] took a different approach, providing a first step to build a general-purpose modeling language for local search. It introduced the concept of *invariants* to express arbitrary one-way constraints that are automatically and incrementally updated under variable assignments. These invariants can then encode, in a declara-

tive fashion, the incremental data structures typically needed in the implementation of meta-heuristics such as Tabu Search [5], Min-Conflict Search [10], Simulated Annealing [6], Scatter Search [7], or GRASP [4] to name just a few.

Constraint-Based Local Search is the culmination of this line of work. It complements the constraint programming efforts typically focused on complete search techniques and delivers a "model and search" framework in which one uses decision variables and constraints to model a problem and relies on search procedures to explore the underlying search space. COMET is an optimization platform that embodies Constraint-Based Local Search and is a direct descendant of LOCALIZER. It is an object-oriented programming language with explicit support for modeling problems declaratively and solving them with local search techniques. The contributions underlying COMET span from the incremental computation model, the modeling abstractions, and the control mechanisms to specify and execute local search heuristics and meta-heuristics easily, efficiently, and compactly.

Foundations

Constraint-Based Local Search aims at implementing the vision captured by the equation

$$\text{LocalSearch} = \text{Model} + \text{Search}$$

that expresses the belief that a local search algorithm is best viewed as the composition of a *declarative* model *with a search* component. This separation of concerns is central: it postulates the importance of expressing the structures of the problem being solved declaratively and compositionally and providing a search component which *exploits* those structures and guide the search toward high-quality local optima.

The Constraint-Based Local Search architecture delivers several key benefits:

Rich Language Constraints are declarative and capture the problem substructures. They range from simple arithmetic constraints, the indexing of arrays of variables with variables, meta-constraints (constraints on the truth value of other constraints) and logical constraints, to combinatorial constraints such as cardinality, sequence, or alldifferent constraints to name just a few. Constraints (and combinators) for local search were introduced in [24].

Rich Search Programming meta-heuristics is supported by a wealth of language combinators and control abstractions to automate the most tedious and error-prone aspects of an actual implementation. The abstractions foster the decoupling of neighborhood, heuristics, and meta-heuristics specifications while leveraging the incrementality exposed by the constraints present in the declarative model. Control abstractions were introduced in [20].

Separation The untangling of model and search promotes the independent design and evolution of these two components. With Constraint-Based Local Search, it is possible to explore alternative models independently of refinements to the search procedures.

Versatility The ability to specify meta-heuristics orthogonally to the model also enables a collection of *generic* search routines which are highly reusable and promote the experimental process to design an effective CBLS program. This versatility further promotes the reuse of highly generic "canned" search procedures.

Extensibility New constraints and objective functions can be added to the system library and used in conjunction with native constraints. Perhaps even more interestingly, the new constraints can be implemented directly in the host language (i.e., COMET), and the bulk of the implementation is often cast in terms of invariants. New heuristics and meta-heuristics can also be easily added to the system as the implementation of *any* heuristic or meta-heuristic relies on a few key concepts such as closures, continuations, selectors, and neighbors.

Performance Finally, the architecture heavily relies on incrementally maintaining the computational state over time. This capability is compositional and a direct by-product of the declarative model. The approach enables the platform to deliver Constraint-Based Local Search *programs* that are a fraction of the size and complexity of handcrafted code, yet deliver performance comparable, and sometime exceeding, manually crafted implementations.

The rest of this chapter starts with an illustration of Constraint-Based Local Search through the modeling and resolution of the classic $n-$queens problem in section "Getting Started." Section "Foundations" discusses the theoretical underpinnings of Constraint-Based Local Search starting with models and concluding with programs. Section "Case Studies" focuses on how to model applications with a rich language that goes beyond Boolean formulas or linear equations. Section "Implementation" explores the implementation issues, starting with the support for incremental computation through invariants and proceeding with a discussion of differentiation. Section "Empirical Results" gives a brief survey of the type of performance that can be expected from Constraint-Based Local Search systems, and section "Conclusion" concludes the chapter.

Getting Started

To introduce Constraint-Based Local Search, it is valuable to start with the modeling and resolution of a simple well-known problem. The examples presented in this chapter uses the COMET platform.

The Problem

The $n-$queens problem is to place n queens on a chess board of size $n \times n$ so that no two queens can attack each other. While the problem is polynomial, its simplicity is appealing to illustrate Constraint-Based Local Search. The COMET program is shown in Fig. 1 and features two clear components that are discussed next.

```
1 import lssolver;
2 int n = 8;
3 range Size = 1..n;
4 Solver<LS> m();
5 ConstraintSystem<LS> S(m);
6 var{int} queen[Size](m,Size);
7 S.add(alldifferent(queen));
8 S.add(alldifferent(all(i in Size) (queen[i] + i)));
9 S.add(alldifferent(all(i in Size) (queen[i] - i)));
10
11 while(S.violations() > 0)
12   selectMin(i in Size,v in Size: queen[i]!=v)(S.getAssignDelta(queen[i],v))
13     queen[i] := v;
```

Fig. 1 A Constraint-Based Local Search model for the queens problem

The Model

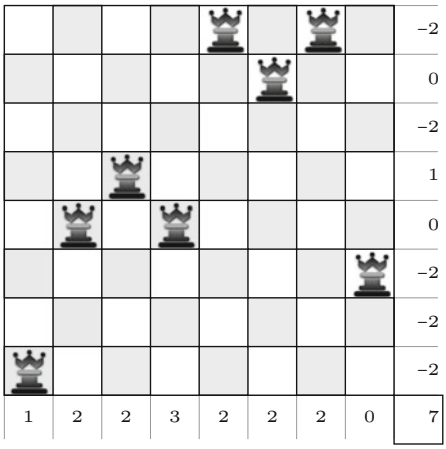
The model definition spans lines 4–9, is declarative, and exclusively focuses on defining an array of decision variables queen in line 6 and three combinatorial constraints on lines 7–9. Variable queen_i models the row on which the queen in column i is to be placed. Each decision variable has a domain $Size$ representing the set of permissible values for that variable. Each combinatorial `alldifferent` constraint takes as input an array of expressions and requires that the values of all entries in the array be pairwise distinct. For instance, line 7 states that no two queens can be assigned to the same row. The constraints on lines 8 and 9 play a similar role but for the upward and downward diagonals. Note how an operator `all` is used on line 8 to create an array of expressions

$$(\text{queen}_1 + 1, \text{queen}_2 + 2, \text{queen}_3 + 3, \cdots, \text{queen}_7 + 7, \text{queen}_8 + 8)$$

which is the input of the combinatorial constraint.

The Search

A classic Constraint-Based Local Search starts from a tentative assignment of values to the decision variables and iteratively transforms it, moving from tentative assignments to tentative assignments using a *local* move operator. The local move used in this specific case is the assignment of a single variable to a new value. Namely, with n decision variables, each with a domain of size n, one could consider up to $\Theta(n^2)$ such moves. To illustrate the search process, consider the left board in Fig. 2a which depicts a tentative assignment. Here, a move consists of relocating a queen to a new row and the algorithm chooses the best such move. The right side, Fig. 2b, shows the reduction in violations for each possible move. Therefore, a viable move is to relocate queen_4 to row 3, reducing the total number of conflicts (violations) from 7 to 5.



(a) Starting point.

0	−1	0	−2	0	0	0	2
1	−1	−1	0	0	0	0	2
−1	−1	−1	−2	0	−1	−1	0
1	0	0	1	0	0	−1	2
1	0	0	0	−1	−1	0	1
1	0	1	−2	0	−1	−1	0
0	−1	−2	−2	−2	−1	−1	0
0	−1	−1	−2	0	0	0	1

(b) Delta Matrix.

Fig. 2 The first step of the CBLS algorithm for the queens problem. (**a**) Starting point. (**b**) Delta matrix

The search outlined above is a classic greedy search. The implementation spanning lines 11–14 obtains the set of constraints present in the model (line 11) and starts from a randomly initialized tentative assignment σ_0. It proceeds through a sequence of moves aimed at satisfying the three softened constraints present in S. The underlying neighborhood function N defined as

$$N(\sigma) = \{\sigma' \mid \exists j \in 1 \dots 8 \; \forall i \in 1 \dots 8, i \neq j : \sigma'(\text{queen}_i) = \sigma(\text{queen}_i) \wedge \\ \sigma'(\text{queen}_j) \in D(\text{queen}_j) \setminus \{\sigma(\text{queen}_j)\}\}$$

is implemented with the `select` statement on line 12 that considers the reassignment of a single decision variable at a time. Overall, the search produces a sequence of tentative assignments

$$\sigma_0, \sigma_1 \in N(\sigma_0), \sigma_2 \in N(\sigma_1), \cdots, \sigma_k \in N(\sigma_{k-1})$$

delivering a solution σ_k that satisfies all the softened constraints (violations of S are 0).

Foundations

This section reviews the foundations of Constraint-Based Local Search, covering both models and programs. It introduces the main concepts behind the declarative and operational models.

Models

From an end-user perspective, constraints and objective functions are at the heart of Constraint-Based Local Search. They provide the declarative bricks needed to construct models describing requirements and properties of the solution being sought. They also support the underlying computational model. This section first reviews the key concepts of constraints and objective functions before finishing with a presentation of the underlying semantics.

Basics

Constraint-Based Local Search is the process of looking for an assignment of values to decision variables that meets specific requirements. Decision variables are central to the modeling process as they characterize solutions. This chapter focuses on integer variables only for simplicity. Extensions to more complex variables (i.e., set, paths, and trees) have been proposed in [12, 23]. The chapter also assumes that Constraint-Based Local Search models are defined over a set X of decision variables.

Definition 1 (Assignment). An assignment σ is a mapping $X \rightarrow D(X)$ from variables to values in their domain. The value assigned to x in σ is denoted by $\sigma(x)$. The set of all possible assignments is denoted by Σ.

For convenience, the expression $\sigma[x/v]$ denotes a new assignment σ' which is similar to σ except that variable x is assigned to v, i.e.,

$$\forall y \in X \setminus \{x\} \; : \; \sigma'(y) = \sigma(y) \wedge \sigma'(x) = v.$$

Constraints are used to impose requirements on the decision variables. Constraints are naturally declarative, can be expressed in a variety of ways, and always capture a relation over a subset of variables from X.

Definition 2 (Constraint). A constraint $c(x_1, \cdots, x_n)$ is a $n-$ary relation over variables $x_1 \cdots x_n \in X$. The set vars(c) is the set of n variables appearing in c, i.e., vars$(c) = \{x_1, \cdots, x_n\}$.

Constraints can be expressed through algebraic expressions, logical statements, or combinatorial structures.

Example 1. In the queens example, the requirement that any two variables cannot lay on the same row, i.e., `alldifferent(x)`, is semantically equivalent to the conjunction of constraints

$$\bigwedge_{i \in 1...8, j \in i+1...8} x_i \neq x_j.$$

Operationally, however, the `alldifferent` maintains its state and violations more efficiently than the naive reformulation above.

Example 2. A constraint system S is a set of constraints and its truth value is equivalent to the truth value of $\wedge_{c \in S} c$.

Evaluations and Violations

The driving force behind Constraint-Based Local Search rests on the ability to assess how badly constraints are violated. This is captured by the concept of violation degrees (or violations for short).

Definition 3 (Violation Degree). The violation degree of $c(x_1, \cdots, x_n)$ is a function $v_c : \Sigma \rightarrow \mathbb{R}^+$ such that, for any assignment σ such that $\sigma(x_1) \in D(x_1), \cdots, \sigma(x_n) \in D(x_n)$, it holds that

$$c(\sigma(x_1), \cdots, \sigma(x_n)) \equiv v_c(\sigma) = 0.$$

The violation degree definition depends critically on the structure conveyed by constraint c. The definition of violation is constraint-dependent but is derived systematically for algebraic and logical expressions. For completeness, consider the specification of expression evaluation.

Definition 4 (Expression Evaluation). Let $e \in \mathcal{E}$ be an arithmetic expression and $\sigma \in \Sigma$ be an assignment. The evaluation of e with respect to (wrt) σ is specified by the function $\mathbb{E}(\sigma, e) : \mathcal{E} \times \Sigma \rightarrow \mathbb{Z}$ which is defined inductively on the structure of e in Fig. 3.

Algebraic *constraints* are based on relations and logical combinators. To reason about the violations of relational and logical constraints, it is useful to derive an *expression* modeling the violations of constraint c from the structure of c.

Definition 5 (Constraint Conversion). Let c be a logical or relational constraint. The constraint conversion of c is an expression $\mathbb{V}(c)$ which can be evaluated to determine the violations of c with respect to some assignment. It is specified by the function $\mathbb{V}(c) : \mathcal{E} \rightarrow \mathcal{E}$ which is defined by induction on the structure of c as specified in Fig. 4.

Fig. 3 The evaluation of an expression with respect to an assignment

$$\begin{aligned}
\mathbb{E}(\sigma, v) &= v \Leftrightarrow v \in \mathbb{R} \\
\mathbb{E}(\sigma, x) &= \sigma(x)(x \in X) \\
\mathbb{E}(\sigma, e_1 + e_2) &= \mathbb{E}(\sigma, e_1) + \mathbb{E}(\sigma, e_2) \\
\mathbb{E}(\sigma, e_1 - e_2) &= \mathbb{E}(\sigma, e_1) - \mathbb{E}(\sigma, e_2) \\
\mathbb{E}(\sigma, e_1 \times e_2) &= \mathbb{E}(\sigma, e_1) \times \mathbb{E}(\sigma, e_2) \\
\mathbb{E}(\sigma, \textsc{Abs}(e)) &= \textsc{Abs}(\mathbb{E}(\sigma, e)) \\
\mathbb{E}(\sigma, \min(e_1, e_2)) &= \min(\mathbb{E}(\sigma, e_1), \mathbb{E}(\sigma, e_2))
\end{aligned}$$

Fig. 4 The constraint conversion function

$$
\begin{aligned}
\mathbb{V}(e_1 = e_2) &= \textsc{Abs}(e_1 - e_2) \\
\mathbb{V}(e_1 \leqslant e_2) &= \max(e_1 - e_2, 0) \\
\mathbb{V}(e_1 \geqslant e_2) &= \max(e_2 - e_1, 0) \\
\mathbb{V}(e_1 \neq e_2) &= 1 - \min(1, \textsc{Abs}(e_1 - e_2)) \\
\mathbb{V}(r_1 \wedge r_2) &= \mathbb{V}(r_1) + \mathbb{V}(r_2) \\
\mathbb{V}(r_1 \vee r_2) &= \min(\mathbb{V}(r_1), \mathbb{V}(r_2)) \\
\mathbb{V}(\neg r) &= 1 - \min(1, \mathbb{V}(r))
\end{aligned}
$$

The conversion for the conjunction of two constraints is none other than the sum of the converted relation violations.

Example 3. The conversion of $x = 5 \wedge y \neq 10$ where $x, y \in X$ is derived as follows:

$$\mathbb{V}(x{=}5 \wedge y \neq 10) {=} \mathbb{V}(x = 5) {+} \mathbb{V}(y \neq 10) = \textsc{Abs}(x{-}5){+}1{-}\min(1, \textsc{Abs}(y{-}10)).$$

It is now possible to compose both functions to obtain the actual violations of an algebraic constraint c with respect to an assignment σ.

Example 4. The violation of an arithmetic constraint $c \equiv l \geq r$ is given by

$$\mathbb{E}(\sigma, \mathbb{V}(l \geq r))$$

The violation degree function for c is then given by

$$v_c(\sigma) = \mathbb{E}(\sigma, \textsc{Abs}(r - l)) = \max(\mathbb{E}(\sigma, r) - \mathbb{E}(\sigma, l), 0).$$

Example 5. The violation of a conjunction $c = c_1 \wedge c_2$ is defined as

$$v_c(\sigma) = \mathbb{E}(\sigma, \mathbb{V}(c_1 \wedge c_2)).$$

While the above mechanics are appropriate to obtain the violations of algebraic constraints, combinatorial constraints define their own notion of violations that capture the combinatorial substructure at hand. Consider the `alldifferent` constraint again.

Example 6. The violation function of $c = \texttt{alldifferent}(x_1, \cdots, x_n)$ simply counts the number of values used more than once by variables $x_1, \ldots, x_n$. More precisely, given an assignment σ and $\text{vars}(c) = \{x_1, \cdots, x_n\}$,

$$v_c(\sigma) = \sum_{i \in S} \max(0, |\{x_j \in \text{vars}(c) | \sigma(x_j) = i\}| - 1)$$

where $S = \bigcup_{i \in 1 \ldots n} D(x_i)$. Consider the `alldifferent` constraint on the rows for the board in Fig. 2a. The assignment is $\sigma = [1, 4, 5, 4, 8, 7, 8, 3]$: It has 2 variables using value 4 and 2 variables using value 8, giving a violation of 2.

Differentiation

The infrastructure covered so far enables the incremental assessment of the satisfaction, and the number of violations, of a constraint c. However, it does not specify how to assess the impact of a local move on the satisfiability or violations of constraints. *Gradients* are the cornerstone of this process.

Definition 6 (Gradient). Given an assignment σ and an arithmetic expression e, $\uparrow_x (\sigma, e)$ denotes the maximum increase for the evaluation of e over all possible values in $D(x)$ wrt σ, whereas $\downarrow_x (\sigma, e)$ denotes the largest decrease for the evaluation of e over all possible values in $D(x)$ wrt σ, i.e.,

$$\uparrow_x (\sigma, e) = \max_{v \in D(x)} \mathbb{E}(\sigma[x/v], e) - \mathbb{E}(\sigma, e)$$

$$\downarrow_x (\sigma, e) = \mathbb{E}(\sigma, e) - \min_{v \in D(x)} \mathbb{E}(\sigma[x/v], e).$$

Note that gradients are nonnegative in this specification. An efficient implementation can be derived inductively on the structure of expression e. Figure 5 gives an abridged version of such derivation. The definition for $\uparrow_x (\sigma, x)$ is an interesting

$$\uparrow_x (\sigma, e) = \max_{v \in D(x)} \mathbb{E}(\sigma[x/v], e) - \mathbb{E}(\sigma, e)$$

$$\downarrow_x (\sigma, e) = \mathbb{E}(\sigma, e) - \min_{v \in D(x)} \mathbb{E}(\sigma[x/v], e).$$

$$\begin{aligned}
&\uparrow_x (\sigma, v) && = 0 \Leftrightarrow v \in \mathbb{R} \\
&\uparrow_x (\sigma, y) && = 0 \Leftrightarrow y \in X \backslash \{x\} \\
&\uparrow_x (\sigma, x) && = \max_{v \in D(x)} v - \sigma(x) \\
&\uparrow_x (\sigma, e_1 + e_2) && = \uparrow_x (\sigma, e_1) + \uparrow_x (\sigma, e_2) \\
&\uparrow_x (\sigma, e_1 - e_2) && = \uparrow_x (\sigma, e_1) + \downarrow_x (\sigma, e_2) \\
&\uparrow_x (\sigma, \textsc{Abs}(e)) && = \max(\textsc{Abs}(\mathbb{E}(\sigma, e) + \uparrow_x (\sigma, e)), \textsc{Abs}(\mathbb{E}(\sigma, e) - \downarrow_x (\sigma, e))) - \mathbb{E}(\sigma, \textsc{Abs}(e)) \\
&\uparrow_x (\sigma, \min(e_1, e_2)) && = \min(\mathbb{E}(\sigma, e_1) + \uparrow_x (\sigma, e_1), \mathbb{E}(\sigma, e_2) + \uparrow_x (\sigma, e_2)) - \mathbb{E}(\sigma, \min(e_1, e_2))
\end{aligned}$$

$$\begin{aligned}
&\downarrow_x (\sigma, v) && = 0 \Leftrightarrow v \in \mathbb{R} \\
&\downarrow_x (\sigma, y) && = 0 \Leftrightarrow y \in X \backslash \{x\} \\
&\downarrow_x (\sigma, x) && = \sigma(x) - \min_{v \in D(x)} v \\
&\downarrow_x (\sigma, e_1 + e_2) && = \downarrow_x (\sigma, e_1) + \downarrow_x (\sigma, e_2) \\
&\downarrow_x (\sigma, e_1 - e_2) && = \downarrow_x (\sigma, e_1) + \uparrow_x (\sigma, e_2) \\
&\downarrow_x (\sigma, \textsc{Abs}(e)) && = \texttt{if } \mathbb{E}(\sigma, e) \geqslant 0 \\
& && \quad \texttt{then } \min(\mathbb{E}(\sigma, e), \downarrow_x (\sigma, e)) \\
& && \quad \texttt{else } \min(-\mathbb{E}(\sigma, e), \uparrow_x (\sigma, e)) \\
&\downarrow_x (\sigma, \min(e_1, e_2)) && = \mathbb{E}(\sigma, \min(e_1, e_2)) - \min(\mathbb{E}(\sigma, e_1) + \downarrow_x (\sigma, e_1), \mathbb{E}(\sigma, e_2) + \downarrow_x (\sigma, e_2))
\end{aligned}$$

Fig. 5 Expression gradients

```
1 interface Constraint<LS> {
2   bool holds();
3   var{int} violations();
4   var{int} violations(var{int} x);
5   int getAssignDelta(var{int} x,int v);
6 }
```

Fig. 6 The constraint interface

base case. Indeed, $\uparrow_x (\sigma, x) = \max_{v \in D(x)} v - \sigma(x)$ which has the effect of picking up the largest increase as the distance between the value currently assigned to x in σ and the largest value of the domain. Similarly, note how $\uparrow_x (\sigma, e_1 - e_2)$ combines the largest increase in e_1 with the largest decrease in e_2.

The next concept, variable violation, is interesting: It captures how many violations can be attributed to a specific variable for a given assignment. Variable violations are specified in terms of gradients.

Definition 7 (Variable Violations). Given a constraint c, the variable violations of c wrt $x \in \text{vars}(c)$ and assignment σ are specified by $\downarrow_x (\sigma, \mathbb{V}(c))$.

The concepts of violation degrees and variable violations are generic and hence they enable the specification of a common API for all constraints. This API makes it possible to implement the slogan

$$\text{LocalSearch} = \text{Model} + \text{Search}$$

mentioned in the introduction. In particular, the API of a constraint is specified in Fig. 6. For instance, the method call `violations()` simply returns the evaluation of $v_c(\sigma)$, while the method call `violations(x)` return $\downarrow_x (\sigma, \mathbb{V}(c))$, for the current variable assignment σ. Finally, the call to method `getAssignDelta(x,v)` returns the variation in violation degree when using the assignment $\sigma[x/v]$ instead of σ, i.e., it returns

$$v_c(\sigma) - v_c(\sigma[x/v]).$$

Combinatorial constraints like `alldifferent` also conform to this interface, and the implementation of the last two methods takes advantage of the constraint semantics to implement the specification incrementally.

Objective Functions

Objective functions play an equally critical role within Constraint-Based Local Search. Objectives provide the necessary mechanics to express and exploit objective functions. Interestingly, once gradients are available, objectives do not add much complexity. Objective functions must conform to the interface depicted in Fig. 7.

```
1 interface Objective<LS> {
2   var{int} evaluation();
3   var{int} increase(var{int} x);
4   var{int} decrease(var{int} x);
5   int getAssignDelta(var{int} x,int v);
6 }
```

Fig. 7 The objective interface

Given the current variable assignment σ, the call `evaluation()` returns $\mathbb{E}(\sigma, e)$, while the call `increase(x)` returns $\uparrow_x (\sigma, e)$ and the call `decrease(x)` returns $\downarrow_x (\sigma, e)$.

Models and Constraint Hardness

From a purely pragmatic and operational standpoint, it is often convenient to partition the actual constraint set C in two

$$C = S \cup R \quad (S \cap R = \emptyset)$$

where R represents a set of *required* but easy to solve constraints and S represents a set of *softened* constraints that are typically much harder to satisfy. The intent is to handle both type of constraints differently. Intuitively, required constraints are always satisfied during the search, while softened constraints may be violated. In the $n-$queens examples introduced earlier, $R = \emptyset$ and all the constraints are softened in S. In general, however, R may contain some constraints that are not worth relaxing in S. The membership in S or R is clearly a design decision for the modeler to consider.

Definition 8 (Constraint Model). A Constraint-Based Local Search model M for a constraint optimization problem is a quintuplet $M = \langle X, D, F, R, S \rangle$ where

- Every $x \in X$ is a decision variable taking its value from $D(x)$,
- F is, without loss of generality, a minimization function,
- R is a set of required constraints (easy to satisfy), and
- S is a set of soft constraints (difficult to satisfy).

Declaratively, the semantics of a Constraint-Based Local Search model is given by the optimization problem

$$\min_{\sigma \in \Sigma} \mathbb{E}(\sigma, F) + \sum_{c \in S} v_c(\sigma)$$

subject to

$$v_c(\sigma) = 0 \quad : \forall c \in R$$

Namely, the objective is to minimize the sum of the violations over the softened constraints and the original objective function subject to the required constraints. In the case of constraint satisfaction, $\mathbb{E}(\sigma, F) = 0$ for every σ as the objective is absent. Note that constraints can be weighted, which makes it possible to balance the two terms of the objective. It is simple to write a combinator to weight a constraint in a generic way [23].

Definition 9 (Feasible Solution). A feasible solution σ to a Constraint-Based Local Search model $M = \langle X, D, F, R, S \rangle$ satisfies

$$\sum_{c \in S} \nu_c(\sigma) = 0.$$

$\mathcal{F}(M)$ denotes the set of feasible solutions to M.

Note that, by definition, every assignment σ also satisfy all the required constraints ($\nu_c(\sigma) = 0 \ : \ \forall c \in R$).

Definition 10 (Optimal Solution). An optimal solution σ^* to a Constraint-Based Local Search model $M = \langle X, D, F, R, S \rangle$ is a feasible solution $\sigma^* \in \mathcal{F}(M)$ such that

$$\forall \sigma \in \mathcal{F}(M) : \mathbb{E}(\sigma^*, F) \leq \mathbb{E}(\sigma, F).$$

Definition 11 (Search Procedure). A search procedure produces a sequence of assignments $\sigma_0, \cdots, \sigma_k$ where $\forall i \in 0..k \ : \ \nu_c(\sigma_i) = 0$ ($c \in R$) and returns σ satisfying

$$\min_{\sigma \in \{\sigma_0, \cdots, \sigma_k\}} \mathbb{E}(\sigma, F) + \sum_{c \in S} \nu_c(\sigma).$$

A Constraint-Based Local Search procedure succeeds if $\sigma \in \mathcal{F}(M)$.

Programs

The model specifies the properties satisfied by assignments appearing in a trace $s_0, s_1, \cdots, s_k$; It does not dictate how the assignments in the trace are generated. This is the prerogative of concrete search procedures which are often viewed as the composition of a *neighborhood* function, a *legality* restriction function, and a candidate *selection* function.

Definition 12 (Neighborhood). The neighborhood of an assignment σ_k, denoted $N(\sigma_k)$, is the set of assignments reachable from σ_k via a local move.

Local moves can be *microscopic* changes to the candidate solution such as the reassignment of a single variable or the swap of the values associated with two distinct variables. In specific domains, e.g., in scheduling, the move can be *macroscopic* and involve changing several variables to capture moves in more complex neighborhood, e.g., moving a task in a job sequence.

Example 7. In the 8−queens example outlined in section "Getting Started", the neighborhood is based on the reassignment of a single queen to a new row. The neighborhood consist of $\Theta(n^2)$ assignments. It could be further restricted to $\Theta(n)$ assignments by only considering the queen appearing in the largest number of conflicts and its possible reassignments. Given a model $M = \langle X, D, 0, \emptyset, S\rangle$, where S is the set consisting of three softened `alldifferent` constraints, the quadratic neighborhood function is

$$N(\sigma_k) = \{\sigma_k[x/v] \mid x \in X \wedge v \in D(x) \setminus \{\sigma_k(x)\}\},$$

while the linear neighborhood function is

$$N(\sigma_k) = \left\{ \sigma_k[x/v] \mid x \in \underset{y \in X}{\text{arg-max}} \downarrow_y (\sigma_k, \mathbb{V}(\underset{c \in S}{\wedge} c)) \wedge v \in D(x) \setminus \{\sigma_k(x)\} \right\}.$$

Definition 13 (Legal Neighbors). The legal neighborhood of an assignment σ_k is a restriction of $N(\sigma_k)$, namely, $L(N(\sigma_k), \sigma_k) \subseteq N(\sigma_k)$, and the legal subset is required to at least satisfy all the required constraints R.

Note that the legal subset of $N(\sigma_k)$ may even be more restrictive and reject neighbors that are feasible with respect to R but fail to exhibit other characteristics. The *full* characterization of the function L is part of the definition of a meta-heuristic. For instance, the tabu meta-heuristic excludes neighbors that were seen in the recent past.

Definition 14 (Neighbor Selection). The selection function S is responsible for choosing the next assignment among legal neighbors. Namely,

$$\sigma_{k+1} = S(L(N(\sigma_k), \sigma_k), \sigma_k) \in L(N(\sigma_k), \sigma_k).$$

For instance, a greedy selection chooses the best neighbor, i.e.,

$$\sigma_{k+1} \in \underset{\sigma \in N(\sigma_k)}{\text{arg-min}}\, \mathbb{E}(\sigma, \mathbb{V}(S))$$

A heuristics or meta-heuristics specifies the three functions N, L, and S which parameterize the computation model.

Control Primitives

To support these abstractions, Constraint-Based Local Search programs rely on a handful of control primitives designed to automate tedious and error-prone implementation details.

While the specification of a program relies on three distinct functions N, L, and S, any implementation concerned with performance produces code that fuses the three functions often degrading the readability and maintainability in the process. Key considerations such as randomization and tiebreaking add another layer of complexity to the code. The implementation of meta-heuristics induces its share of complexity by refining move legality and selection further.

COMET attempts to strike a delicate balance between efficiency, code readability, and ease of maintenance by decoupling these aspects as much as possible. The language introduces *selectors*, *neighbors*, and *randomized choosers* as control abstractions.

Neighborhood Selectors

The concept of *neighborhood selector* is a cornerstone for the specification of complex local searches. A simplified version of its interface is presented below:

```
1 interface Neighborhood {
2   void insert(int q,Closure c);
3   boolean hasMove();
4   Closure getMove();
5 }
```

The key idea is that a neighbor is defined as a pair $\langle q, c\rangle$ consisting of a quality measure q and a closure c. The intuition is that, in general, the closure c defines the move which, when executed, will produce an objective value of q for the resulting assignment. Method `insert` adds a neighbor c of quality q (line 2), method `hasMove` checks whether the neighborhood is nonempty, and method `getMove` returns the selected move.

Concrete implementations of this interface commit to a specific selection policy. For instance, the concrete selector `MinNeighborSelector` implements the neighborhood interface and retains only a neighbor minimizing the quality measure. Namely, if the set of inserted neighbors is $N = \{\langle q_1, c_1\rangle, \cdots, \langle q_n, c_n\rangle\}$, the selector retains *one* neighbor

$$\langle q, c\rangle \in \underset{\langle q_j, c_j\rangle \in N}{\text{arg-min}}\, q_j$$

and produces it as its selection when `getMove` is called. Alternative selectors include `KMinNeighborSelector` that returns one of the top$-k$ best neighbors (k being a parameter of the selector).

```
1 MinNeighborSelector N();
2 while(S.violations()!=0) {
3    forall(i in Size,v in queen[i].getDomain() : queen[i] != v)
4       neighbor(S.getAssignDelta(queen[i],v),N) queen[i] := v;
5    if (N.hasMove()) call(N.getMove());
6 }
```

Fig. 8 An alternative greedy search for $n-$queens

The `neighbor` Control Primitive

Neighborhood selectors work in conjunction with the `neighbor` control primitive. The primitive has the following syntax:

`neighbor`$(\langle expr \rangle, \langle N \rangle)$ $\langle Body \rangle$

where $\langle expr \rangle$ refers to an arbitrary arithmetic expression, $\langle N \rangle$ is an expression referring to a selector, and $\langle Body \rangle$ is a statement (or block of statements). From a semantics standpoint, the control primitive creates a closure c of the body code responsible for producing the assignment σ_{k+1} from the current assignment σ_k. It then associates the closure with a quality measure q and submits the pair $\langle q, c \rangle$ to the selector denoted by N. What makes `neighbor` particularly attractive is the syntactic closeness between the code specifying the quality of the move and the code carrying out the move even if, in practice, there is a strong temporal disconnection between the time when the closure is recorded with the selector and the time it gets executed.

The search procedure in Fig. 8 illustrates an alternative implementation of the greedy search presented in Fig. 1. The loop on lines 3–4 scans all variables and values accumulating the reassignments in the selector N and associating with each one a quality measure based on the delta. The 1-instruction block `queen[i] := v;` on line 4 is automatically wrapped in a closure that is entrusted by `neighbor` to the selector N. The beauty in the construction lies in the ability to easily accumulate in the same selector the union of multiple neighborhoods, all defined with different move operators. This capability will be illustrated in the Progressive Party Problem use case in section "Progressive Party."

Randomized Choosers

To support developers, COMET provides control abstractions to make greedy, semi-greedy, and randomized choices. The abstractions encapsulate the necessary state to deliver independent pseudorandom streams in each such instruction appearing in the program. For instance, the $\Theta(n)$-sized neighborhood that considers reassigning the queen with the most violations to a new row, is easily modeled with instructions in lines 3–7 below.

```
1 MinNeighborSelector N();
2 while(S.violations()!=0) {
3     selectMax[2](i in Size)(S.violations(queen[i])) {
4         forall(v in queen[i].getDomain() : queen[i] != v)
5             neighbor(S.getAssignDelta(queen[i],v),N)
6                 queen[i] := v;
7     }
8     if (N.hasMove()) call(N.getMove());
9 }
```

The bracketed 2 on line 3 simply requests the selector to retain any value i among the best two (according to the violation measure).

Discussion

The Constraint-Based Local Search model appearing in Fig. 1 highlights the key features of the framework. It features a complete separation between a declarative model and a procedural search component. The declarative model relies on combinatorial constraints specifying the properties of solutions, while the search is exclusively devoted to the selection of a heuristic and meta-heuristic. In this model, the search remains problem specific. Yet, both the model and the search can evolve independently. One can add constraints without changing the search or choose a different search strategy without modifying the model. All these characteristics, when blended with the performance of an incremental computation, deliver an appealing architecture for producing local search solutions.

The program in Fig. 1 can still be improved. In particular, it is tempting to provide syntactic sugar to automatically handle the boiler plate code and support the notion of *constraint annotations* to partition the constraint set into soft and required constraints $S \cup R$. The resulting program is shown in Fig. 9. The model m is now attaching a *soft* annotation to each constraint to dictate its addition to S, while line 10 is used to extract the set of soft constraints from m. Perhaps even more interestingly, the adoption of an explicit first-class model concept enables the authoring of completely generic search procedures. Indeed, the code in lines 10–13 can be packaged as the reusable min-conflict procedure depicted in Fig. 10.

Naturally, lines 10–13 from Fig. 9 disappear from the model in favor of a single line calling the generic `minConflictSearch` on model m program, i.e.,

```
1 minConflictSearch(m);
```

This leaves us with a 10 lines Constraint-Based Local Search implementation.

```
MinNeighborSelector N();
while(S.violations()!=0) {
   selectMax[2](i in Size)(S.violations(queen[i])) {
      forall(v in queen[i].getDomain() : queen[i] != v)
         neighbor(S.getAssignDelta(queen[i],v),N)
            queen[i] := v;
   }
   if (N.hasMove()) call(N.getMove());
}
```

```
import lssolver;
int n = 8;
range Size = 1..n;
model m {
   var{int} queen[Size](Size);
   soft: alldifferent(queen);
   soft: alldifferent(all(i in Size) (queen[i] + i));
   soft: alldifferent(all(i in Size) (queen[i] - i));
}
ConstraintSystem<LS> S = m.getSoftConstraintSystem();
while(S.violations() > 0)
  selectMin(i in Size,v in Size : queen[i]!=v)(S.getAssignDelta(queen[i],v))
    queen[i] := v;
```

Fig. 9 A revised Constraint-Based Local Search model for n-queens

```
void function minConflictSearch(Model<LS> m) {
   ConstraintSystem<LS> S = m.getSoftConstraintSystem();
   var{int}[] X = S.getIntVariables();
   while(S.violations() > 0)
     selectMin(i in X.getRange(),
               v in X[i].getDomain() : X[i]!=v)(S.getAssignDelta(X[i],v))
       X[i] := v;
}
```

Fig. 10 A reusable min-conflict procedure

Case Studies

To explore Constraint-Based Local Search, it is desirable to consider a few applications that are elegantly and effectively solved with COMET. The next three subsections consider the progressive party problem [16], car sequencing [3], and scene allocation [18] as each application illustrates a different aspect.

Progressive Party

The progressive party problem is a standard benchmark in combinatorial optimization and it illustrates two important features. It shows a rich model with many combinatorial constraints as well as constraints on the truth value of relations. It also shows how soft constraints may be instrumental in obtaining a neighborhood. The goal in this problem is to assign guest parties to boats (the hosts) over multiple time periods. Each guest can visit the same boat only once and can meet every other guest at most once over the course of the party. Moreover, for each time period, the guest assignment must satisfy the capacity constraints of the boats.

Figure 11 depicts the declarative part of the model. The decision variable `boat[g,p]` specifies the boat visited by guest `g` in period `p`. Lines 8–9 specify the alldifferent constraints for each guest, lines 10–11 specify the capacity constraints, and lines 12–13 state that two guests meet at most once during the evening. The `soft(2)` annotations added on lines 9 and 11 are not only specifying that the constraint must be softened, but they also associate a fixed static weight of 2 with each constraint. The weights can be easily incorporated in the inductive definition of $\mathbb{V}(c)$ shown in Fig. 4

$$\mathbb{V}(\texttt{soft}(w) : c) = w \cdot \mathbb{V}(c)$$

to handle *statically weighted* soft constraints.

The Neighborhood

The first sensible neighborhood to consider focuses on reassigning a single variable `boat[g,p]` to a new value. This can follow the same template used for the queens problem. These moves impact the violations of *all* the constraints. However, these moves may also prove too restrictive. When an instance is near satisfaction, one can expect most of the bins in any given knapsack to be near full. A single

```
1  range Hosts = 1..13;
2  range Guests = 1..29;
3  range Periods = 1..up;
4  set{int} config[1..6];
5
6  model m {
7     var{int} boat[Guests,Periods](Hosts);
8     forall(g in Guests)
9        soft(2): alldifferent(all(p in Periods) boat[g,p]);
10    forall(p in Periods)
11       soft(2): knapsack(all(g in Guests) boat[g,p],crew,cap);
12    forall(i in Guests, j in Guests : j > i)
13       soft: atmost(1,all(p in Periods) boat[i,p] == boat[j,p]);
14 }
```

Fig. 11 The progressive party problem

variable reassignment amounts to moving an item from one bin into another and that may prove fruitless from a violation standpoint. A natural idea is to include a *second neighborhood* that considers the exchange of values between two variables appearing in the same constraint. While such swaps have no effects on the `alldifferent` constraints, they can more easily lead to violation improvements for the `knapsack` constraints. Formally, the neighborhood is therefore

$$\begin{aligned} N = \{\sigma[x/v] \quad & | \; x = \text{arg-max}_{z \in X} \downarrow_z (\sigma, \mathbb{V}(S)) \wedge v \in D(x) \setminus \{\sigma(x)\}\} \bigcup \\ \{\sigma[x/c, y/d] \; & | \; x = \text{arg-max}_{z \in X} \downarrow_z (\sigma, \mathbb{V}(S)) \wedge \\ & \quad y \in \bigcup_{c \in S \wedge x \in \text{vars}(c)} \text{vars}(c) \wedge c = \sigma(y) \wedge d = \sigma(x)\} \end{aligned}$$

where S is the set of soft constraints.

The code in Fig. 12 implements that idea. The main loop (lines 7–30) seeks to improve the overall violations of the soft constraints. The selector on line 9 picks

```
1  int tenure = 2;
2  int it = 0;
3  int tabu[Guests,Periods,Hosts] = 0;
4
5  ConstraintSystem<LS> S = m.getSoftConstraintSystem();
6  MinNeighborSelector N();
7  while (S.violations() > 0) {
8     int old = S.violations();
9     selectMax(g in Guests, p in Periods)(S.violations(boat[g,p])) {
10       forall(h in Hosts : tabu[g,p,h] <= it) {
11          neighbor(S.getAssignDelta(boat[g,p],h),N) {
12             tabu[g,p,boat[g,p]] = it + tenure;
13             boat[g,p] := h;
14          }
15       }
16       selectMin(g1 in Guests,d=S.getSwapDelta(boat[g,p],boat[g1,p]))(d) {
17          neighbor(d,N) {
18             tabu[g,p,boat[g1,p]] = it + tenure;
19             tabu[g1,p,boat[g,p]] = it + tenure;
20             boat[g,p] :=: boat[g1,p];
21          }
22       }
23    }
24    if (N.hasMove()) {
25       call(N.getMove());
26       if (violations < old && tenure > 2) tenure = tenure - 1;
27       if (violations >= old && tenure < 10) tenure = tenure + 1;
28    }
29    it = it + 1;
30 }
```

Fig. 12 The search procedure for the progressive party problem

a variable `boat[g,p]` that induces the most violations. Once the variable is selected, the two *nested* selectors (lines 10–15 and lines 16–22) implement the two parts of the neighborhood structure. Lines 10–15 select the variable with the most violations and choose a new value that decreases its violations the most. The second selector is more interesting. It picks a second variable `boat[g1,p]` in the same period `p` as the first variable `boat[g,p]` in such a way that the swap between the two variables has the largest impact on the overall violations of S. Note that both variables appear in the knapsack constraint stated over period p. The remainder of the code (lines 23–28) executes the best move in N (if one exist) and updates the variable tabu tenure.

The 30 lines of code are compact and elegant: They benefit from the automation provided by the neighborhood selector, the selectors, and the neighbor construction. Yet, they still require some effort to analyze the model and produce a code template that follows a standard recipe. In addition, this code skeleton must still be updated with classic ideas like search intensification around high-quality local minima and diversification to escape from basins of attraction. However, the steps applied in deriving this search procedure are rather systematic: They rely on an analysis of the model to recognize the presence of specific types of constraints that then suggest a particular neighborhood structure. The is the idea behind the synthesis of search procedures that is described next.

The Synthesized Search

Given that models are first-class objects, COMET can manipulate and analyze them. Here, COMET recognizes the presence of knapsack constraints and generates a *composite* neighborhood consisting of the union of variable assignments and of the variable swaps appearing in violated knapsack constraints, an idea first articulated by Van Hentenryck [19]. The synthesis process per se was described in details in [22].

The skeleton of a synthesized tabu search is depicted in Fig. 13. It uses the fact that all variables have the same domains. Lines 7–9 associate with each decision variable x_i the set of constraints mentioning x_i and susceptible to benefit from exchanging (swapping) the values of two of its variables. This initial step is a straightforward projection of the constraint array in S that only retains constraints that refer to x_i and *can use a swap*. Line 10 defines a neighborhood selector. Line 15 selects the variable x_i with the most violations among the softened constraints S. Lines 16–20 focus on the first part of the neighborhood and consider simple reassignments that lead to the largest violation decrease. All the potential neighbors are submitted to N. Lines 21–29 consider all the constraints referring to variable x_i and for which swaps can be of potential benefit. For each such constraint, the code retrieves the variables of the constraint and accumulates in the selector N all the closures modeling swaps (along with their impact on violations).

It is appealing to notice the similarities between the manual and synthetic implementation. In essence, they both capture the same idea. Yet, the generic code adapts to the model and consider exploiting all the constraints susceptible to profit

```
1  function bool refersTo(var{int}[] av,var{int} x) {
2     return or(i in av.rng()) (av[i].getId() == x.getId());
3  }
4  function void minConflictWithSwap(Model<LS> m) {
5     ConstraintSystem<LS> S = m.getSoftConstraintSystem();
6     var{int}[] X = S.getVariables();
7     set{Constraint<LS>} cx[i in X.rng()] =
8                         collect(j in S.rng() : S[j].canUseSwap() &&
9                                 refersTo(S[j].getVariables(),X[i])) S[j];
10    MinNeighborSelector N();
11    int k = 0,tenure = 20;
12    int at[X.rng()] = 0;
13    int mat[X.rng(),X.rng()] = 0;
14    forall(int k=0;S.violations() != 0;k++) {
15       selectMax(i in X.getRange())(S.violations(X[i])) {
16          forall(v in X[i].getDomain() : at[i] <= k)
17             neighbor(S.getAssignDelta(X[i],v),N) {
18                X[i] := v;
19                at[i] = k + tenure;
20             }
21          forall(c in cx[i]) {
22             var{int}[] Y = c.getVariables();
23             forall(j in Y.getRange() : mat[i,j] <= k)
24                neighbor(S.getSwapDelta(X[i],Y[j]),N) {
25                   X[i] :=: Y[j];
26                   mat[i,j] = mat[j,i] = k + tenure;
27                }
28          }
29       }
30       if (N.hasMove()) call(N.getMove());
31    }
32 }
```

Fig. 13 The synthesized search procedure for the progressive party problem

from swaps. The COMET extension required to do so is a minimal extension to query the capabilities of the constraint in the model.

As shown later, the synthesized search outperforms all published results on this problem. Note that, if the set of required constraints R was not empty (it is empty in this application), the two neighborhoods would have to discard assignments and swaps that yield nonzero values for the calls to `getAssignDelta` and `getSwapDelta` to implement the legality requirement correctly. Finally, observe that this skeleton implementation contains a very simply tabu condition to further restrict the legal moves to those that were not recently attempted; this is achieved with two simple data structures (one per neighborhood) that record the last iteration number when a move was performed. Supporting generic intensification and diversification is equally easy.

Car Sequencing

Figure 14 presents a model for car sequencing. In this application, n cars must be sequenced on an assembly line of length n. The customer demands for car configurations are specified in an array $demand$ and the total demand is, of course, n. Each car configuration may require a different sets of options, while capacity constraints on the production units restrict the possible car sequences. For a given option o, these constraints are of the form k out of m meaning that, out of m successive cars, at most k can require o. The model declares the decision variables specifying which type of car is assigned to each slot in the assembly line (line 12). It states a hard constraint specifying which cars must be produced (line 13) and then states the soft capacity constraints for each option (lines 14–15).

The handcrafted search procedure, illustrated in Fig. 15, is modeled after a conflict minimization structure. The initialization satisfies the cardinality constraint by using a random permutation of an array of values that already meet the cardinality requirement (lines 4–7). The main loop (lines 10–27) minimizes the number of violations by selecting the slot of the assembly line causing the most violations and swapping its content with another slot that delivers the most improvements. The move itself appears on line 14. The search features a *diversification* component (lines 19–25) that randomly swaps a subset of slots in the assembly line when no improvement took place for a number of iterations. Each time an improving move is found, the stability counter is reset and the best value for this stage is recorded (line 18).

It is possible to recognize the combinatorial structure present in the model thanks to the presence of global `sequence` constraints and to automatically synthesize a search procedure that matches the ideas present in the handcrafted search procedure.

```
1 // ... read parameters nbCars, nbConfigs, nbOptions
2 range Cars = 1..nbCars;
3 range Configs = 0..nbConfigs-1;
4 range Options = 1..nbOptions;
5 boolean requires[Configs,Options];
6 int demand[Configs];
7 int lb[Options],ub[Options];
8 // ... read the data ...
9
10 set{int} options[o in Options] = setof(c in Configs) requires[c,o];
11 model m {
12     var{int} line[Cars](Configs);
13     hard: atmost(demand,line);
14     forall(o in Options)
15         soft: sequence(line,options[o],lb[o],ub[o]);
16 }
```

Fig. 14 The car sequencing model

```
1  ConstraintSystem<LS> S = m.getSoftConstraintSystem();
2  Solver<LS> ls = m.getLocalSolver();
3  int best = System.getMAXINT();
4  int cars[Cars];
5  int nb = 0;
6  forall(c in Configs, n in 1..demand[c])
7      cars[++nb] = c;
8  RandomPermutation perm(Cars);
9  forall(c in Cars) lines[c] := cars[perm.get()]; // Satisfy required constraint
10 while (S.violations() != 0) {
11     selectMax(i in Cars)(S.violations(X[i])) {
12         selectMin(j in Cars : line[i] != line[j] && t[i,j] <= it)
13             (S.getSwapDelta(line[i],line[j])) {
14             X[i] :=: X[j];
15             t[i,j] = t[j,i] = it + tenure;
16         }
17     }
18     if (S.violations() < best) { best = S.violations();stable = 0;}
19     if (stable == 500) {
20         with atomic(ls)
21             forall(k in 1..5)
22                 select(a in Cars,b in Cars : line[a] != line[b]) line[a] :=: line[b];
23         stable := 0;
24         best = S.violations();
25     } else stable++;
26     it++;
27 }
```

Fig. 15 The handcrafted search procedure for car sequencing

The result is shown in Fig. 16. It depicts a generic tabu search procedure featuring several key components. Note how line 6 of the model retrieves the required constraints (line 3) and initializes the start assignment by delegating to each required constraint the task of picking a suitable assignment. A basic requirement for achieving this is the following independence property of the required constraints:

$$\forall c_i, c_j \in R \text{ s.t. } i \neq j : \text{vars}(c_i) \cap \text{vars}(c_j) = \emptyset.$$

In this case, $R = \{\texttt{atmost(demand,line)}\}$ and the sole cardinality constraint can create, in polynomial time, an array of values that meet the cardinality requirement and randomly permute it to produce the initial assignment to the variables in vars(c). The algorithm for the cardinality constraint uses a (randomized) feasible flow algorithm to match values to variables. As all variables appear in the cardinality constraint, swapping two variables is feasibility-preserving (the number of cars of each type is unchanged). Moreover, the cardinality constraint is tight, meaning that there is a bijection from variable to value occurrences. The core of the

```
1  function void tabuSearch(Model<LS> m) {
2     ConstraintSystem<LS> S = m.getSoftConstraintSystem();
3     ConstraintSystem<LS> R = m.getHardConstraintSystem();
4     Solver<LS> ls = m.getLocalSolver();
5     var{int}[] X = R.getVariables();
6     forall(i in R.getRange()) with atomic(ls) R.getConstraint(i).initialize();
7     Counter it(ls) := 0;
8     Counter stable(ls) := 0;
9     int tenure = 20;
10    int t[X.rng(),X.rng()] = 0;
11    Integer best(System.getMAXINT());
12    whenever S.violations()@changes(int o,int n)
13       if (n < best) { best := n;stable := 0;}
14    whenever it@changes() stable++;
15    whenever stable@changes()
16       if (stable == 500) {
17          with atomic(ls)
18             forall(k in 1..5)
19                select(a in X.rng(),b in X.rng() : X[a] != X[b]) X[a] :=: X[b];
20          stable := 0;
21       }
22    while (S.violations() != 0) {
23       selectMax(i in X.rng())(S.violations(X[i])) {
24          selectMin(j in X.rng() : X[i] != X[j] && t[i,j] <= it)
25             (S.getSwapDelta(X[i],X[j])) {
26             X[i] :=: X[j];
27             t[i,j] = t[j,i] = it + tenure;
28          }
29       }
30       it++;
31    }
32 }
```

Fig. 16 The synthesized Tabu-search for car sequencing

search spans lines 22–31 and features the selection of the most conflicting variable (line 23) together with the variable that yield the largest decrease in violations through a swap (lines 24–25). The actual move is performed on line 26 and the move is marked tabu in line 27.

The implementation of the diversification is more interesting. To modularize the capability, it relies on *events*. In COMET, objects like `Counter`, `Integer`, or `var{int}` are capable of dispatching *notifications* when specific events occur. For instance, a counter issues a *change* event when its value is modified. COMET provides the ability to associate a code fragment with events: The code is then executed in response to these events. This is illustrated on line 14 that states that, each time the iteration counter `it` changes, the stability counter `stable` must increase. Similarly, lines 15–21 specify that, when the stability counter changes, one should check whether it has reached a critical value, 500 in this example,

in which case a diversification step is undertaken. This architecture promotes the separation of the diversification logic from the main heuristic. Indeed, the code for the diversification simply dictates how to react to changes to the stability counter and the COMET platform automatically *weaves* that code in the proper place. Finally, note how recording improvements in the violations are also done through an event hooked on the violations of the entire soft system S.

Scene Allocation

Figure 17 features a model for the scene allocation problem that is sometimes used to compare CP and MIP solvers since it is highly symmetric. The problem consists of assigning specific days for shooting scenes in a movie. There can be at most five scenes shot per day and all actors of a scene must be present. Each actor has a fee and is paid for each day she/he plays in a scene. The goal is to minimize the production cost. The decision variable $scene$ (line 12) represents the day a scene is shot. The hard cardinality constraint (line 13) specifies the maximum number of scenes shot on any one day. The objective function minimizes the sum of actor compensations, which is the actor fee times the number of days he/she appears in a scene shot on that day.

The model in Fig. 17 is now a constraint optimization model featuring an objective function that demands a different strategy to obtain a suitable search procedure. In general, it is necessary to juggle three considerations. First, one should maintain the feasibility of the required constraints through a suitable initialization and the selection of moves that never violate the constraints in R. Second, one

```
1  int maxScene = ...; // read the data
2  int maxDay = ...; // read the data
3  range Scenes = 1..maxScene;
4  range Days = 1..maxDay;
5  enum Actor = ... ; // read the data
6  int pay[Actor] = ...; // read the data
7  set{Actor} appears[Scenes] = ...; // read the data
8  set{int} which[a in Actor] = setof(s in Scenes) member(a,appears[s]);
9  int occur[Days] = ...; // read the data
10
11 model m {
12    var{int} scene[Scenes](Days);
13    hard: atmost(occur,scene);
14    minimize: sum(a in Actor) pay[a]*
15                    (sum(d in Days) (or(s in which[a]) (scene[s] == d)));
16 }
17 generatedTabuSearch(m);
```

Fig. 17 Scene allocation model

may have to deal with softened difficult constraints (S) for whom one searches for a feasible solution by driving down the violations. Third, it is essential to drive down the value of the objective function. The latter two considerations (true objective and softened constraint) are naturally conflicting as it is easier to drive the objective function down if one violates difficult constraints and vice versa. The practical response is to rely on a statically or a dynamically weighted sum of the two objectives where the search shifts the emphasis on either considerations by altering the weights.

In this application, $S = \emptyset$ and $R = \{\texttt{atmost}(occur, scene)\}$; hence, the task is somewhat simpler as there is only one component to the objective function. In this case, the generated tabu search is driven by the sole required constraint but with a significant difference. In the previous example, the soft constraint was tight, a fact detected by the model analysis. Indeed, the cardinality constraint in car sequencing is tight because the assembly line has as many slots as the number of cars to produce. This is not the case here: there are typically fewer scenes than the number of slots in which they can be scheduled and thus the required constraint is not a bijection between variables and value occurrences. As a result, limiting the neighborhood only to swaps would preserve the feasibility of the cardinality constraint at the expense of a significant decrease in solution quality. This is not surprising since the neighborhood would no longer be connected. The model analysis however recognizes that the `atmost` constraint is not tight and also considers feasibility-preserving assignments.

The generated skeleton is depicted in Fig. 18. As before, line 5 uses the required constraints to initialize the search to a feasible assignment (once again, the task is delegated to the required constraints and the model analysis ensures that $\text{vars}(c_i) \cap \text{vars}(c_j) = \emptyset \, \forall c_i, c_j \in R$ before generating code. The core of the search spans lines 11–24 and relies on the union of two neighborhoods. Line 12 starts by selecting a variable x_i that can lead to the largest decrease in the objective function. Lines 13–15 consider all the swaps that include x_i and lead to the largest decrease in the objective function (i.e., the most negative delta). The selector on line 13 is semi-greedy and will select, uniformly at random, one of the top-3 such moves. Lines 16–19 are devoted to the second neighborhood and collect the best value to reassign x_i. Line 17 shows the conjunct that eliminates assignments that are not feasible with respect to R. The skeleton search uses a vanilla tabu data structure and omits the diversification component for simplicity.

Implementation

The implementation of a Constraint-Based Local Search system critically depends on incremental computation. Constraints and objective functions must respond to their APIs like `violations`, `increase`, `decrease`, or `getAssignDelta` extremely fast in order to consider large neighborhoods and long traces of assignment within an allotted time.

```
function void generatedTabuSearch(Model<LS> m) {
   Function<LS> obj = m.getObjective();
   ConstraintSystem<LS> R = m.getHardConstraintSystem();
   Solver<LS> ls = m.getLocalSolver();
   forall(i in R.getRange()) with atomic(ls) R.getConstraint(i).initialize();
   int it = 0,tenure = 20,best = System.getMAXINT();
   var{int}[] X = obj.getVariables();
   int tm[X.rng(),X.rng()] = 0;
   int t[X.rng()] = 0;
   MinNeighborSelector N();
   while (it < 10000) {
      selectMax(i in X.rng())(obj.decrease(X[i])) {
         selectMin[3](j in X.rng() : i != j && tm[i,j] <= it,
                      d = obj.getSwapDelta(X[i],X[j]))(d)
            neighbor(d,N) { X[i] :=: X[j];tm[i,j] = tm[j,i] = it + tenure;}
         selectMin(v in X[i].getDomain() : t[i] <= it && v != X[i] &&
                                           R.getAssignDelta(X[i],v)==0,
                   d = obj.getAssignDelta(X[i],v))(d)
            neighbor(d,N) { X[i] := v;t[i] = it + tenure;}
      }
      if (N.hasMove()) call(N.getMove());
      if (obj.evaluation() < best) best = obj.evaluation();
      it++;
   }
}
```

Fig. 18 Generated search for scene allocation

To deliver this performance, the implementation is presented in two distinct layers. The `invariant` layer is responsible for the basic incremental computation that occurs when an assignment is changed through a local move operator. The `differentiability` layer is responsible for implementing the response mechanism behind the constraints and objective function. Their implementation is primarily framed in terms of invariants. Both are highlighted in this section, starting with invariants (section "Invariants") and finishing with differentiation (section "Differentiation").

Invariants

Invariants provide a declarative concept that relieves programmers from the tedious task of maintaining complex data structures incrementally. By focusing on what to maintain, rather than how to maintain assignments under changes, programmers are relieved of an error-prone, yet critical, aspect of implementing constraints and objective functions. In essence, invariants capture so-called *one-way constraints* [1, 2, 11, 17], namely, they capture the value of an expression that must be

maintained over time under changes to the value of its variables. An acyclic network of dependencies connects all the variables of the problems and is responsible for scheduling the evaluations. This subsection reviews examples and outlines the underlying implementation.

Definition 15 (Invariant). An invariant $\mathcal{I}$ is a one-way constraint

$$\langle x_1, \ldots, x_n \rangle \leftarrow f(y_1, \ldots, y_m)$$

where $x_1, \ldots, x_n$ are called invariant variables and $y_1, \ldots, y_m$ are either decision or invariant variables. The set $O = \{x_1, \ldots, x_n\}$ is the output variables of $\mathcal{I}$ The set $I = \{y_1, \ldots, y_m\}$ are the input variables of $\mathcal{I}$. We often abuse notation and rewrite the one-way constraints as

$$O \leftarrow f(I).$$

$\mathcal{I}_O$, $\mathcal{I}_I$, and $\mathcal{I}_f$ denote the output variables, the input variables, and the function of invariant $\mathcal{I}$.

Since invariants are one-way constraints, there are some necessary syntactic restrictions. One such restriction is that no two invariants have a common output variable. This ensures that every invariant variable is defined at most once.

The declarative semantics of an invariant specifies that the one-way constraints always holds.

Definition 16 (Declarative Semantics of an Invariant). Let σ be an assignment before or after any atomic instruction of a program for which the invariant $\mathcal{I}$ has been posted. Then, it follows that

$$\langle \sigma(x_1), \ldots, \sigma(x_n) \rangle \leftarrow f(\sigma(y_1), \ldots, \sigma(y_m))$$

where $\mathcal{I}_O = \{x_1, \ldots, x_n\}$ and $\mathcal{I}_I = \{y_1, \ldots, y_m\}$. The narrative abuses notation and sometimes writes

$$\sigma(\mathcal{I}_o) \leftarrow \mathcal{I}_f(\sigma(\mathcal{I}_I)).$$

Example 8 (Expression Invariant). The numerical invariant

$$x \leftarrow y + 3 * z$$

stating that, at any point in time, the value of x in an assignment σ, i.e., $\sigma(x)$, should be the value $\sigma(y)$ plus three times the value of $\sigma(z)$.

Operationally, an invariant must maintain the link between its output and input variables under assignments to its input variables. The invariant maintains this link

by keeping a local assignment of its input variables and by implementing an update function which updates the global and local assignments to reflect the change in one of its input variables.

Definition 17 (Operational Semantics of an Invariant). Let $\mathcal{I}$ be an invariant. Operationally, $\mathcal{I}$ maintains a local store $\mathcal{I}_\sigma$ over variables $\mathcal{I}_I$ and implements an update procedure $\mathcal{I}_u : \Sigma \times X$. Let $\sigma_l = \mathcal{I}_\sigma$ be the local store of $\mathcal{I}$, σ be an assignment, and $y \in \mathcal{I}_I$. The update procedure $\mathcal{I}_u(\sigma, y)$ performs the following assignments:

$$\sigma(\mathcal{I}_O) := \mathcal{I}_f(\sigma_l[\sigma(y)/y](\mathcal{I}_I));$$
$$\sigma_l(y) := \sigma(y);$$

To propagate a collection of invariants effectively, the implementation constructs an incremental graph.

Definition 18 (Incremental Graph). An incremental graph $G(X \cup I, A)$ is a directed acyclic graph whose vertices coincides with decision variables and invariants and whose arc set correspond to the dependencies induced by the invariant. Each invariant $\mathcal{I}$ introduces a dependency $\mathcal{I} \leftarrow y$ for each $y \in \mathcal{I}_I$ and a dependency $x \leftarrow \mathcal{I}$ for each $x \in \mathcal{I}_O$.

Figure 19 depicts the dependencies of the arithmetic invariant presented earlier.

Example 9 (Summation Aggregate). A summation invariant captures the relation

$$x \leftarrow \sum_{i=0}^{n-1} y[i]$$

where n is a constant and y denotes an array of n variables. The dependencies are shown below:

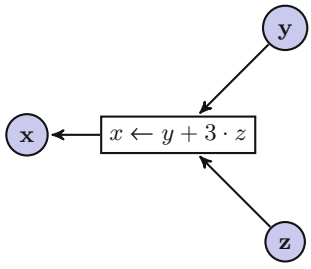


Fig. 19 The dependencies of an arithmetic invariant

The update function $\mathcal{I}_u(\sigma, y)$ implements the following code:

```
1 σ(x) := σ(x) + σ(y) − σ_l(y);
2 σ_l(y) := σ(y);
```

where $\sigma_l = \mathcal{I}_\sigma$.

Example 10 (Counting). A counting invariant $x \leftarrow \text{count}(y)$ defined over an array of variables y yields an array of variables x indexed by $R = \bigcup_{i=0}^{n-1} D(y_i)$ that maintains the relations

$$\forall\, v \in R \,:\, x_v = \sum_{i \,\in\, \text{range}(y)} (y_i = v)$$

Namely, x_v counts the number of variables in y currently assigned to v. The dependencies are as follows:

and there are $|R| - 1 + n$ of them. The update function $\mathcal{I}_u(\sigma, y)$ implements the following code:

```
1  σ(x_{σ_l(y)}) := σ(x_{σ_l(y)}) − 1;
2  σ(x_{σ(y)}) := σ(x_{σ(y)}) + 1;
3  σ_l(y) := σ(y);
```

where $\sigma_l = \mathcal{I}_\sigma$.

Incremental Computation

Given $G(X \cup I, A)$, one can obtain a topological sort r of its vertices. Indeed, each dependency $y \leftarrow x$ imposes the constraint

$$1 + r(x) \leq r(y).$$

The partial ordering expressed in r drives the propagation algorithm that updates all the variables following an assignment of new values to decision variables.

Figure 20 shows the pseudo-code for the invariant propagation algorithm. The `propagate` algorithm is invoked with the incremental graph $G(X, A)$, an assignment σ and the set of decision variables Y that have been updated. Line 3 initializes a priority queue PQ with all the invariants mentioning any member of Y as one of its sources. The priority associated with invariant $\mathcal{I}$ is its topological number $r(\mathcal{I})$. The main loop spanning lines 6–11 considers the invariants in priority order (see Line 7). The update function of the selected invariant is executed in Line 8. Line 9 collects in C the modified output variables and Line 10 enqueues the new invariant to reconsider.

The correctness of the `propagate` algorithm hinges on the facts that $G(X, A)$ is acyclic. The use of topological numbers guarantee that the invariant considered in iteration i is handled only after its sources have reached final values in σ. As long as u meets its specification, the assignment σ is guaranteed to satisfy all the

```
1 propagate(G(X, A),σ,Y)
2 {
3     PQ = ⋃_{y∈Y} {⟨I ← y, r(I)⟩ ∈ A};
4     while (PQ ≠ ∅) {
5         σ_o := σ;
6         ⟨I ← y, r(I)⟩ := extractMin(PQ);
7         I_u(σ, y);
8         C = {x ∈ I_O : σ_o(x) ≠ σ(x)};
9         forall(x ∈ C)
10            PQ = PQ ∪ {⟨I ← x, r(I)⟩ ∈ A};
11    }
12 }
```

Fig. 20 The invariant propagation

invariants considered in iterations $1 \dots i$. Computing the affected variables in C and scheduling any invariant depending on them cannot possibly schedule an earlier invariant since the graph is acyclic.

Differentiation

The implementation of constraints and objective functions relies on the foundation provided by invariants as first described in [21]. As indicated earlier, the implementation of the constraint API

```
1 interface Constraint<LS> {
2   bool holds();
3   var{int} violations();
4   var{int} violations(var{int} x);
5   int getAssignDelta(var{int} x,int v);
6 }
```

depends on an efficient, incremental evaluation of violations, variable violations, and gradients. The functions $\mathbb{E}(\sigma, e)$, $\mathbb{V}(e)$, $\uparrow_x (\sigma, e)$, and $\downarrow_x (\sigma, e)$ are essential to the evaluation of expressions and the definition of violation and gradient expressions from algebraic definitions. Yet, none of them are incremental and therefore unsuitable for *direct* use in, for instance, Example 5. Likewise, this is true for combinatorial constraints such as `alldifferent`. Invariants do provide the solution, and the subsection focuses on the implementation when constraints are expressed algebraically or combinatorially.

Algebraic and Logical Constraints

The key insight to an incremental implementation is to forsake the evaluation function $\mathbb{E}$ and adopt instead a *compilation* approach, generating invariants that evaluate an expression incrementally. This is achieved by Function $\mathbb{I} : \mathcal{E} \rightarrow X \times 2^I$

Fig. 21 Compiling expression evaluations to invariants

$$
\begin{array}{ll}
\mathbb{I}(v) & = \langle i_v, \{i_v \leftarrow v \Leftrightarrow v \in \mathbb{R}\}\rangle \\
\mathbb{I}(x) & = \langle i_x, \{i_x \leftarrow x\}\rangle \\
\mathbb{I}(e_1 + e_2) & = \text{let } \langle i_1, S_1\rangle = \mathbb{I}(e_1), \langle i_2, S_2\rangle = \mathbb{I}(e_2) \\
 & \quad \text{in } \langle i_+, \{i_+ \leftarrow i_1 + i_2\} \cup S_1 \cup S_2\rangle \\
\mathbb{I}(e_1 - e_2) & = \text{let } \langle i_1, S_1\rangle = \mathbb{I}(e_1), \langle i_2, S_2\rangle = \mathbb{I}(e_2) \\
 & \quad \text{in } \langle i_-, \{i_- \leftarrow i_1 - i_2\} \cup S_1 \cup S_2\rangle \\
\mathbb{I}(e_1 \times e_2) & = \text{let } \langle i_1, S_1\rangle = \mathbb{I}(e_1), \langle i_2, S_2\rangle = \mathbb{I}(e_2) \\
 & \quad \text{in } \langle i_\times, \{i_\times \leftarrow i_1 \times i_2\} \cup S_1 \cup S_2\rangle \\
\mathbb{I}(\textsc{Abs}(e)) & = \text{let } \langle i_e, S_e\rangle = \mathbb{I}(e) \\
 & \quad \text{in } \langle i_{abs}, \{i_{abs} \leftarrow \textsc{Abs}(i_e)\} \cup S_e\rangle \\
\mathbb{I}(\min(e_1, e_2)) & = \text{let } \langle i_1, S_1\rangle = \mathbb{I}(e_1), \langle i_2, S_2\rangle = \mathbb{I}(e_2) \\
 & \quad \text{in } \langle i_{min}, \{i_{min} \leftarrow min(i_1, i_2)\} \cup S_1 \cup S_2\rangle \\
\mathbb{I}(\sum_{k=0}^{n} e_k) & = \text{let } \langle i_k, S_k\rangle = \mathbb{I}(e_k) \ \forall\ k\ \in 0..n \\
 & \quad \text{in } \langle i_\Sigma, \{i_\Sigma \leftarrow \sum_{k=0}^{n} i_k\} \cup \bigcup_{k=0}^{n} S_k\rangle
\end{array}
$$

(which is partly shown in Fig. 21) and defined inductively on the structure of expressions. Specifically, a call $\mathbb{I}(e)$ on an expression e produces a decision variable holding the value of the expression and a set of invariants that define this variable.

Note how each line of the inductive definition obtains the variable and invariants supporting the operand and produces a fresh variable alongside an additional invariant based on the variables obtained from the inductive calls on the operands. For instance, the last line of Fig. 21 shows that the compilation of a summation expression inductively obtains a fresh decision variable i_k for each term e_k, as well as the invariants supporting i_k's definition in S_k. It then creates a new fresh variable i_Σ and the summation aggregate invariant that defines it. It finally adds all the invariants in S_k.

Relations simply give rise to arithmetic expressions through the function $\mathbb{V}$ whose definition is in Fig. 3. To obtain an incremental evaluation of the violations of an arbitrary relation $e_1 \diamond e_2$, one can simply obtain the violation expression and compile it with:

$$\langle i_{e_1 \diamond e_2}, S_{e_1 \diamond e_2}\rangle = \mathbb{I}(\mathbb{V}(e_1 \diamond e_2))$$

to retrieve a set of invariants (which it states) and an output variable $i_{e_1 \diamond e_2}$ whose value $\sigma(i_{e_1 \diamond e_2})$ denotes the violations of $e_1 \diamond e_2$ with respect to σ. At this point, the implementation of method `violation()` is straightforward and reduces to returning $\sigma(i_{e_1 \diamond e_2})$. The incremental evaluation of gradients proceeds similarly with the generation of an expression modeling the gradient of e and its compilation with $\mathbb{I}$, i.e.,

$$\langle i_{\downarrow_x(e)}, S_{\downarrow_x(e)}\rangle = \mathbb{I}(\downarrow_x (e))$$

and $\downarrow_x (e)$ is an expression (independent of σ) whose evaluation w.r.t. σ would yield $\downarrow_x (\sigma, e)$. Similarly, a method call `violations(x)` must simply return $\sigma(i_{\downarrow_x(e)})$.

Finally, objective functions make a direct use of expressions as well as $\uparrow_x$ and $\downarrow_x$ and are therefore handled exactly like constraints.

Combinatorial Constraints

While combinatorial constraints could be implemented in terms of expressions, it is often preferable to exploit the semantics of the constraints to directly produce an incremental implementation. To illustrate the idea, consider the `alldifferent(x)` constraint used in the introductory example and responsible for ensuring that no two entries in x have the same value.

Fundamentally, the constraint should maintain the cardinality of each value used in x and require that no two values have a cardinality larger than 1 to satisfy the constraint. The variable violation for x_i would, in this case, simply be the excess in the number of variables assigned to $\sigma(x_i)$. In essence, when the constraint `alldifferent(x)` is added on an array x with n variables where $\bigcup_{j=1}^{n} D(x_j) = V$, it is sufficient to state the following invariants:

$$\begin{aligned} c &\leftarrow \text{count}(x) \\ v_i &\leftarrow \max(0, c_i - 1) \;\; \forall i \in V \\ cv &\leftarrow \textstyle\sum_{k \in V} v_i \\ vv_j &\leftarrow v_{x_j} \qquad\qquad \forall j \in 1 \ldots n. \end{aligned}$$

The implementation of the constraint then reduces to

```
1  class alldifferent implements Constraint<LS> {
2    bool holds() { return σ(cv) == 0;}
3    var{int} violations() { return cv;}
4    var{int} violations(var{int} x) { return vv_id(x);}
5    int getAssignDelta(var{int} x,int v) {
6      if σ(x) == v
7        return 0;
8      else
9        return (σ(c_v) ≥ 1) − (σ(x) ≥ 2);
10   }
11 }
```

where the function `id` is used to identify the variable x by an integer. Note the simplicity of the method implementations that simply leverage the work done by invariants. Additionally, the implementation does not have to provide any imperative code to handle the changes of decision variables as all of this logic is handled through the invariants. Finally, even the `getAssignDelta(x,v)` implementation remains straightforward. When the current value assignment to variable x is identical to the tentative assignment (v), the function returns 0. Otherwise, it returns the amount of change. Namely, if value v is already used once or more, the number of violations will increase by 1. Similarly, if $\sigma(x)$ is used twice or more, a violation is necessarily lost.

Empirical Results

Offering a comprehensive empirical evaluation of COMET is beyond the scope of this chapter. Yet, the monograph [23] contains an extensive empirical evaluation on many problems and discusses the impact of modeling techniques and search.

Instead, this section focuses on demonstrating the potential behind the synthesis of search procedures. In particular, it explores the performance of the search procedures shown in Figs. 13, 16, and 18 and contrasts them with the results obtained from purely synthesized search procedures in the spirit of [22]. In all cases, the results were obtained on a Core i7 machine clocked at 1.8 Ghz and running OSX 10.10 and the reported results are based on averages collected from 100 runs of each algorithms.

Progressive Party

Two instances of the problem (5–7 and 6–7) were used in the evaluation in the following table.

Type	Choice	$\|P\|$	μ_{iter}	σ_{iter}	μ_T (msec.)	σ_T (msec.)
Manual	5	7	2,360.9	1,386.1	723.9	377.3
Synthetic	5	7	2,282.2	1,495.9	842.5	476.1
Manual	6	7	6,847.8	5,714.7	1,682.0	1,324.4
Synthetic	6	7	4,532.2	4,724.5	1,338.1	1,281.6

The search procedures are extremely similar and only differ in the presence of an adaptive tabu list within the synthetic implementation. It is not surprising to note that both implementation are very close with a slight win for the synthetic search without having to invest any effort in parameter tuning.

Car Sequencing

One instance (4–72) was used for car sequencing. In this case, the two implementations seem to exhibit significantly different behaviors as the number of iterations is almost 4 times as high (on average) for the manual implementation. This is most likely due to the difference in parameters and in the components of the meta-heuristics within the synthetic search. It also shows that, without exploring variants of the search procedure, it is not obvious to produce high-quality results. Yet, the ability to easily exploit, without further ado, restarts, diversification and intensification components is quite valuable. Lastly, note that the time per iteration of the two searches is quite close showing that the differences only come from the search heuristic and meta-heuristic, not the incremental computation.

Type	Instance	μ_{iter}	σ_{iter}	$\mu_T(s.)$	$\sigma_T(s.)$
Manual	4–72	162,944.3	164,853.9	19.9	20.5
Synthetic	4–72	49,598.8	57,815.4	5.4	6.3

Scene Allocation

The scene allocation benchmark was used to compare three variants: the manual implementation in Fig. 18 and two synthetic search procedures with different parameters, namely, one of them uses 10,000 iterations and no restarts, while the other uses 10 restarts and 1000 iterations per restart. Here, the synthetic search relying on restarting is very close to the custom implementation. Its success rate is 99/100, just shy of a perfect 100 like the custom search. While the synthetic search uses more iterations (on average) to get to the optimum, the runtimes are very close. An examination of the synthesized search reveals that the root cause of the difference is the search heuristic. The equivalent of lines 13 and 17 in Fig. 18 in the synthesized search rely on a purely random selector rather than the more aggressive semi-greedy and greedy selectors used in the custom implementation. This difference explains the loss in greediness and explains the positive impact of restarts.

Type	$\mu_{\text{iter}}*$	$\sigma_{\text{iter}}*$	μ_{f*}	σ_{f*}	#Opt	μ_T(msec.)	σ_T(msec.)
Manual	860	982	334,144	0	100	567.4	46.9
Synthetic(10,000 iters, 1 restart)	2,254	3,104	334,960	1,093	64	731.8	175.6
Synthetic(1000 iters, 10 restarts)	1,954	2,212	334,167	227	99	730.1	71.7

Conclusion

Constraint-Based Local Search is an appealing framework for the design and implementation of local search models for any number of applications. It adopts the core practices of constraint programming through the support of separated components for expressing a declarative model and for programming the search. The declarative models rely on a rich language seamlessly blending algebraic, logical, and combinatorial constraints. The search component itself is exclusively devoted to the automation of the most tedious and error-prone activities that arise when implementing a variety of heuristics and meta-heuristics. Perhaps even more crucially, search procedures can be written completely independently of the model, making them highly reusable and generic. The culmination of this effort is the availability of synthetic search procedures that take advantage of an analysis of the declarative model to produce sensible search procedures that often compete with tailored, hand-written implementations.

The entire framework competitiveness relies on incremental implementations that are constructed on top of invariants and differentiable abstractions such as constraints and objectives. The net result is a platform for practitioners who would take advantage of the capabilities of local search techniques without the significant investment necessary to produce an efficient implementation.

Cross-References

- Ant Colony Optimization: A Component-Wise Overview
- Guided Local Search
- Iterated Greedy
- Restart Strategies
- Scatter Search
- Tabu Search
- Theory of Local Search
- Variable Neighborhood Descent
- Variable Neighborhood Search

References

1. Borning A (1981) The programming language aspects of thinglab, a constraint-oriented simulation laboratory. ACM Trans Program Lang Syst 3(4):353–387
2. Borning A, Duisberg R (1986) Constraint-based tools for building user interfaces. ACM Trans Comput Graph 5(4):345–374
3. Dincbas M, Simonis H, Van Hentenryck P (1988) Solving the car sequencing problem in constraint logic programming. In: ECAI-88, Aug 1988
4. Feo T, Resende M (1995) Greedy randomized adaptive search procedures. J Glob Optim 6:109–133
5. Glover F, Laguna M (1997) Tabu search. Kluwer Academic Publishers, Boston/Dordrecht/London
6. Kirkpatrick S, Gelatt C, Vecchi M (1983) Optimization by simulated annealing. Science 220:671–680
7. Laguna M (2002) Scatter search. In: Pardalos PM, Resende MGC (eds) Handbook of applied optimization. Oxford University Press, New York, pp 183–193
8. Michel L (1998) Localizer: a modeling language for local search. PhD thesis, Brown University
9. Michel L, Van Hentenryck P (1997) Localizer: a modeling language for local search. In: Third international conference on the principles and practice of constraint programming (CP'97), Lintz, Oct 1997
10. Minton S, Johnston MD, Philips AB (1990) Solving large-scale constraint satisfaction and scheduling problems using a Heuristic repair method. In: AAAI-90, Aug 1990
11. Myers B, Guise D, Dannenberg R, Vander Zanden B, Kosbie D, Pervin E, Mickish A, Marchal P (1990) GARNET: comprehensive support for graphical, highly interactive user interfaces. IEEE Comput 23(11):71–85
12. Pham Q-D, Deville Y, Van Hentenryck P (2012) Ls(graph): a constraint-based local search for constraint optimization on trees and paths. Constraints 17(4):357–408
13. Selman B, Kautz H (1993) An empirical study of greedy local search for satisfiability testing. In: AAAI-93, pp 46–51

14. Selman B, Levesque H, Mitchell D (1992) A new method for solving hard satisfiability problems. In: AAAI-92, pp 440–446
15. Selman B, Kautz H, Cohen B (1996) Local search strategies for satisfiability testing. In: DIMACS series in discrete mathematics and theoretical computer science, vol 26. American Mathematical Society Publications. DIMACS
16. Smith BM, Brailsford SC, Hubbard PM, Williams HP (1996) The progressive party problem: integer linear programming and constraint programming compared. Constraints 1:119–138
17. Sutherland IE (1963) SKETCHPAD: a man-machine graphical communication system. MIT Lincoln Labs, Cambridge
18. Van Hentenryck P (2002) Constraint and integer programming in OPL. Inform J Comput 14(4):345–372
19. Van Hentenryck P (2006) Constraint programming as declarative algorithmics. ACP award for research excellence in constraint programming. Available at http://www.cs.brown.edu/people/pvh/acp.pdf
20. Van Hentenryck P, Michel L (2005) Control abstractions for local search. Constraints 10(2):137–157
21. Van Hentenryck P, Michel L (2006) Differentiable invariants. In: 12th international conference on principles and practice of constraint programming (CP'06), Nantes, Sept 2006. Lecture notes in computer science
22. Van Hentenryck P, Michel L (2007) Synthesis of constraint-based local search algorithms from high-level models. In: Proceedings of the 22nd national conference on artificial intelligence – volume 1, AAAI'07. AAAI Press, pp 273–278
23. Van Hentenryck P, Michel L (2009) Constraint-based local search. The MIT Press, Cambridge
24. Van Hentenryck P, Michel L, Liu L (2005) Constraint-based combinators for local search. Constraints 10(3):363–384
25. Walser JP (1999) Integer optimization by local search: a domain-independent approach. Springer, Berlin/Heidelberg. ISBN:3-540-66367-3. http://www.springer.com/us/book/9783540663676