



Constraint-Based Combinators for Local Search

PASCAL VAN HENTENRYCK
Brown University, Box 1910, Providence, RI 02912, USA

pvh@cs.brown.edu

LAURENT MICHEL AND LIYUAN LIU
University of Connecticut, Storrs, CT 06269-3155, USA

ldm@engr.uconn.edu

Abstract. One of the most appealing features of constraint programming is its rich constraint language for expressing combinatorial optimization problems. This paper demonstrates that traditional combinators from constraint programming have natural counterparts for local search, although their underlying computational model is radically different. In particular, the paper shows that constraint combinators, such as logical and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects. These combinators naturally support elegant and efficient modelings, generic search procedures, and partial constraint satisfaction techniques for local search. Experimental results on a variety of applications demonstrate the expressiveness and the practicability of the combinators.

Keywords: constraints language, local search, logical and cardinality combinators, differential objects, first-class expressions, reification, search procedure

1. Introduction

Historically, most research on modeling and programming tools for combinatorial optimization has focused on systematic search, which is at the core of branch & bound and constraint satisfaction algorithms. However, in recent years, increased attention has been devoted to the design and implementation of programming tools for local search (e.g., [3, 10, 11, 15, 22, 27, 29]). This is motivated by the orthogonal strengths of the paradigms, the difficulty of obtaining efficient implementations, and the lack of compositionality and reuse in local search.

COMET [12, 25] is a novel, object-oriented, programming language specifically designed to simplify the implementation of local search algorithms. Comet supports a constraint-based architecture for local search organized around two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the search heuristic and meta-heuristic. Constraints and objective functions are natural vehicles to express combinatorial optimization problems and often capture combinatorial substructures arising in many practical applications. But constraints and objective functions have a fundamentally different computational model in COMET as they do not prune the search space. Rather they are *differentiable objects* that maintain a number of properties incrementally and provide algorithms to evaluate the effect of various operations on these properties. The search component then uses these functionalities to guide the local search using selectors and other high-level control structures [25]. The architecture enables local search algorithms to be high-level, compositional, and modular.

However, constraint programming languages and libraries also offer rich languages for combining constraints, including logical and cardinality operators [24], reification, and expressions over variables. These “combinators” are fundamental in practice, not only because they simplify problem modeling, but also because they lay the foundations for expressing various kinds of ad-hoc constraints that typically arise in complex applications. In addition, these combinators make it possible to write generic search procedures that can be used across a wide range of applications.

This paper shows that traditional constraint programming combinators bring similar benefits for local search, although their underlying computational model differs significantly. In particular, the paper shows that arithmetic, logical, and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects, providing high-level and efficient abstractions for composing constraints and objectives in local search. These combinators naturally support very high-level models, partial constraint satisfaction techniques, and generic search procedures that are independent of the application at hand and only rely on generic interfaces for constraints and objective functions. As a consequence, they foster the separation of concerns between modeling and search components, increase modularity, and favor compositionality and reuse. More generally, the paper shows that the rich language of constraint programming is conceptually robust and brings similar benefits to constraint programming and local search, despite their fundamentally different computational models.

The rest of this paper is organized as follows. Sections 2 and 3 discuss how to combine constraints and objective functions in COMET, Section 4 introduces first-class expressions, and Section 5 discusses generic search procedures. Section 6 briefly discusses modeling issues for local search and contrast them to constraint programming. Section 7 concludes the paper. Applications and experimental results are presented together with the abstractions in these sections.

2. Constraints

Constraints in COMET are differentiable objects that implement the same interface. This section starts by specifying and illustrating the constraint interface, before moving to the presentation of the constraint-based combinators.

2.1. The Constraint Interface

Constraints in COMET are differentiable objects. Given a variable assignment α , a constraint c maintains its truth value $\tau_\alpha(c)$, its violation degree $v_\alpha(c)$, and the variable violations, $v_\alpha(x, c)$ that may be associated with each variable x in c . Constraints also implement differentiable methods to evaluate the effect of moves on their violations.

More precisely, constraints in COMET implement the interface (partially) described in Figure 1. The interface gives access to the constraint variables (method `getVariables`) and to two decision variables which represent the truth value of the constraint (method `isTrue`) and its violation degree (method `violations`). The violation

```

interface Constraint {
    var{int}[] getVariables();
    var{bool} isTrue();
    var{int} violations();
    int getViolations(var{int} x);
    int getAssignDelta(var{int} x,int v);
    int getSwapDelta(var{int} x1,var{int} x2);
    int getAssignDelta(var{int}[] x,int[] v);
}

```

Figure 1. The constraints interface in COMET (partial description).

degree is constraint-dependent and measures how much the constraint is violated. This information is often more useful than the truth value in guiding the local search. There are many ways to define the violations of a constraint. For instance, the violations of an arithmetic constraint $l \geq 1$ is typically given by $\max(0, r - l)$. However, for combinatorial constraints, different concepts of violations have been proposed. See, for instance, [17, 26]. The method `getViolations` returns the violations which may be attributed to a given variable, which is often useful in selecting local moves in some search procedures.

The remaining methods provide the differentiable API of the constraint. They make it possible to evaluate the effect of an assignment, a swap, or multiple assignments on the violation degree. Method `getAssignDelta` can be used to evaluate the impact of assigning value d to variable, x i.e.,

$$\Delta_{\alpha}(x \leftarrow d, c) = v_{\alpha[x \leftarrow d]}(c) - v_{\alpha}(c)$$

where α is the current assignment and $\alpha[x \leftarrow v]$ denotes the assignment α in which x has been assigned to v . Method `getSwapDelta` can be used to evaluate the impact of swapping variables x and y , i.e.,

$$\Delta_{\alpha}(x \leftrightarrow y, c) = v_{\alpha[x \leftrightarrow y]}(c) - v_{\alpha}(c)$$

where α is the current assignment and $\alpha[x \leftrightarrow y]$ denotes the assignment α in which x has been swapped with y .

Example 1 (AllDifferent). Consider the constraint $c = \text{alldifferent}(a)$ that holds if all variables in a are given different values. In COMET, this constraint uses value-based violations:

$$v_{\alpha}(c) = \sum_{d \in D} \max(\#_{\alpha}(d, a) - 1, 0)$$

where $\#_{\alpha}(d, a)$ denotes the number of occurrences of value d in array a under assignment α and D is the domain of all variables in a . The variable violations are once again value-based:

$$v_{\alpha}(x, c) = \max(\#_{\alpha}(\alpha(x), a) - 1, 0)$$

where $\alpha(x)$ denotes the value of variable x under assignment α . The differentiable methods can be implemented in constant time. For instance,

$$\Delta(x \leftarrow d, c) = \begin{cases} 0 & \text{if } \alpha(x) = d; \\ (\#_{\alpha}(d, a) \geq 1) - (\#_{\alpha}(\alpha(x), a) \geq 2) & \text{otherwise.} \end{cases}$$

Maintaining the violations incrementally is performed in time $O(|\delta|)$ where δ is the set of reassigned variables by using the distribute invariant of LOCALIZER [11] and incremental updates of arithmetic sums.

Example 2. (AllPresent). Consider the constraint $c = \text{allpresent}(R, a)$ that holds if all values in R are assigned to some variable in a . The violations of $\text{allpresent}(R, a)$ are once again value-based:

$$v_{\alpha}(c) = \#\{d \in R \mid \#_{\alpha}(d, a) = 0\}$$

The variable violations are defined as

$$v_{\alpha}(x, c) = \#_{\alpha}(\alpha(x), a) > 1$$

indicating that variable x can be reassigned to reduce the violations of c . Selecting a variable x such that

$$\#_{\alpha}(\alpha(x), a) = 1$$

cannot decrease the violations. Once again, observe that these expressions can be maintained efficiently under changes to the variables.

We are now ready to discuss constraint-based combinators, which also are constraints in COMET. Their semantics is summarized in Table 1 for simplicity.

Table 1. The semantics of some constraint combinators

Combinators	Violation Degrees
$c_1 \ \&\& \ c_2$	$v(c_1) + v(c_2)$
$c_1 \ c_2$	$\min(v(c_1), v(c_2))$
$\tau(c)$	If $v(c_1) > 0$ then 1 else 0
exactly $(k, [c_1, \dots, c_n])$	$\text{abs}(\sum_{i=1}^n \tau(c_i) - k)$
atmost $(k, [c_1, \dots, c_n])$	$\max(\sum_{i=1}^n \tau(c_i) - k, 0)$
$k \times c$	$k \times v(c)$

2.2. Constraint Systems

Constraint systems, a fundamental modeling abstraction in COMET, are container objects representing a conjunction of constraints. Constraint systems are constraints themselves and implement the `Constraint` interface. Hence they maintain their truth value and their violation degree, i.e., the sum of the violation degrees of their constraints. They also support the differentiable API. Figure 2 depicts a simple COMET program for the n-queens problem. Lines 1–4 describe the problem variables and data structures, lines 5–9 describe the modeling component, and lines 10–13 specify the search procedure. Line 3 creates a uniform distribution and line 4 declares the decision variables: variable `queen[i]` represents the row of the queen placed in column `i`. These decision variables are initialized randomly using the uniform distribution.¹ Line 5 declares the constraint system and lines 6–8 specify the problem constraints using the ubiquitous *alldifferent* constraint. More precisely, they specify that the queens cannot be placed on the same row, the same upper diagonal, and the same lower diagonal. Lines 10–13 describe a min-conflict search procedure [14]. Line 11 selects the queen `q` with the most violations and line 12 chooses a new value `v` for queen `q`. Line 13 assigns this new value to the queen, which has the effect of (possibly) updating the violation degree of the subset of affected constraints and of the constraint system. Lines 11–13 are iterated until a solution is found.

Constraint systems provide a clean separation between the declarative and search components of a local search. Observe that it is possible to add new constraints to the constraint system without changing the search procedure. Similarly, it is possible to change the search procedure (e.g., adding a tabu list) without modifying the model. It is also important to stress that a single COMET program may use several constraint systems simultaneously. This is useful, for instance, for local search algorithms that maintain the feasibility of a subset of constraints (e.g., the hard constraints), while allowing others to be violated (e.g., the soft constraints).

```

1.  range Size = 1..1024;
2.  LocalSolver m();
3.  UniformDistribution distr(Size);
4.  var{int} queen[i in Size](m,Size) := distr.get();

5.  ConstraintSystem S(m);
6.    S.post(alldifferent(queen));
7.    S.post(alldifferent(all(i in Size) queen[i] + i));
8.    S.post(alldifferent(all(i in Size) queen[i] - i));
9.  m.close();

10. while (S.violations() > 0)
11.   selectMax(q in Size)(S.getViolations(queen[q]))
12.   selectMin(v in Size)(S.getAssignDelta(queen[q],v))
13.   queen[q] := v;

```

Figure 2. A COMET program for the queens problem.

2.3. Logical and Cardinality Operators

One of the appealing features of constraint programming is its ability to combine constraints using logical and cardinality operators. COMET offers similar functionalities for local search. For instance

```
Constraint c = ((x != y) || (x != z));
```

illustrates a disjunctive constraint in COMET. *The disjunctive constraint is a differentiable object implementing the Constraint interface.* In particular, the violation degree of a disjunction $c = c_1 || c_2$ is given by $\min(v(c_1), v(c_2))$, where $v(c)$ denotes the violation degree of c .

COMET also features a variety of cardinality operators. For instance, Figure 3 illustrates the cardinality operator $exactly(k, [c_1, \dots, c_n])$, a differentiable constraint which holds if exactly k constraints hold in c_1, \dots, c_n . The figure depicts a COMET program to solve the magic series problem, a traditional benchmark in constraint programming. A series (s_0, \dots, s_n) is magic if s_i represents the number of occurrences of i in (s_0, \dots, s_n) . Lines 6–10 in Figure 3 specify the modeling component. Line 8 features the cardinality operator to express that there are `magic[v]` occurrences of v in the magic series and line 9 adds the traditional redundant constraint. Lines 11–19 implement a min-conflict search with a simple tabu-search component. Observe the modeling component in this program which is similar to a traditional constraint programming solution. Interestingly, local search performs reasonably well on this problem as indicated in Table 2. The table

```

1. int n = 400;
2. range Size = 0..n-1;
3. LocalSolver m();
4. var{int} magic[i in Size](m,Size) := 0;
5. int tabu[i in Size] = -1;

6. ConstraintSystem S(m);
7. forall(v in Size)
8.   S.post(exactly(magic[v],all(i in Size) magic[i] == v));
9. S.post(sum(i in Size) i * magic[i] == n);
10. m.close();

11. int it = 0;
12. while (S.violations() > 0) {
13.   selectMax(s in Size:tabu[s]<it)(S.getViolations(magic[s]))
14.   selectMin(v in Size:magic[s]!=v)
15.     (S.getAssignDelta(magic[s],v)) {
16.     magic[s] := v;
17.     tabu[s] = it + 3;
18.   }
19.   it = it + 1;

```

Figure 3. A COMET program for the magic series problem.

Table 2. Performance results on the magic series program

n	70	90	110	130	150	170	190	210
$best(T)$	0.21	0.41	0.57	0.84	1.09	1.44	2.09	3.20
$\mu(T)$	1.05	1.78	5.70	13.58	21.70	47.60	67.83	150.41
$worst(T)$	7.35	10.85	30.95	102.63	86.54	347.77	400.20	761.11
$\sigma(T)$	1.73	2.84	9.12	22.87	31.38	81.80	110.71	240.85

gives the best, average, and worst times in seconds for 50 runs on a 2.4 Ghz Pentium, as well as the standard deviation.

The contributions here are twofold. On the one hand, COMET naturally accommodates logical and cardinality operators as differentiable objects, allowing very similar modelings for constraint programming and local search. On the other hand, implementations of logical and cardinality operators directly exploit incremental algorithms for the constraints they combine, providing compositionality both at the language and implementation level. The implementations can in fact be shown optimal in terms of the input/output incremental model [20], assuming optimality of the incremental algorithms for the composed constraints. For instance, the implementation of the combinator

$$atmost(k, [c_1, \dots, c_n])$$

updates its violation $\sum_{i=1}^n \tau(c_i)$ in constant time whenever the truth value of a constraint c_i changes. This is optimal if the composed constraints are optimal and independent. If there are some dependencies between the constraints, it may be possible to derive more efficient algorithms exploiting the underlying structures.²

2.4. Weighted Constraints

Many local search algorithms (e.g., [6, 21, 28]) use weights to focus the search on some subsets of constraints. COMET supports weight specifications which can be either static or dynamic. (Dynamic weights vary during the search). Weights can be specified with the `*` operator. For instance the snippet

```
Constraint c = 2 * alldifferent(x)
```

associates a constant weight of 2 to an *alldifferent* constraint and returns an object implementing the `Constraint` interface. Its main effect is to modify the violation degree of the constraint and the results of its differentiable API. Weights can also be specified by variables, which is useful in many applications where weights are updated after each local search iteration. This feature is illustrated later in the paper in a frequency allocation application.

Figure 4 depicts the modeling component of a COMET program to solve the progressive party problem. It illustrates both constant weights and the cardinality operator *atmost*. Line 7 expresses weighted *alldifferent* constraints to ensure that a party never visits the

```

1. LocalSolver m();
2. UniformDistribution distr(Hosts);
3. var{int} boat[Guests,Periods](m,Hosts) := distr.get();
4. int tabu[Guests,Periods,Hosts] = -1;

5. ConstraintSystem S(m);
6. forall(g in Guests)
7.   S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
8. forall(p in Periods)
9.   S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
10. forall(i in Guests, j in Guests : j > i)
11.   S.post(atmost(1,all(p in Periods) boat[i,p]==boat[j,p]));
12. m.close();

```

Figure 4. The COMET model for the progressive party problem.

same boat twice. Line 9 posts weighted knapsack constraints to satisfy the capacity constraint on the boats. Finally, line 11 uses a cardinality constraint to specify that no two parties meet more than once over the course of the event. The cardinality operator makes for a very elegant modeling: it removes the need for the rather specific *meetAtmostOnce* constraint used in earlier version of the COMET program [12]. It also indicates the ubiquity of cardinality constraints for expressing, concisely and naturally, complex constraints arising in practical applications.

Table 3 describes experimental results for this modeling and the search procedure of [12] augmented with a restarting component that resets the current solution to a random configuration every 100,000 iterations in order to eliminate outlier runs.³ The table describes results for various configurations of hosts and various numbers of periods. The results report the average running times over 50 runs of the algorithm, as well as the best times in parenthesis. With the addition of a restarting component, the standard deviation for the hard instances is always quite low (e.g., configuration 1–9,16–19 with 7 periods has a mean of 95.4 and a deviation of 4.8) and shows that the algorithm’s behavior is quite robust. The results were obtained on a 2.4 Ghz Pentium 4 running Linux. The performance of the program is excellent, as the cardinality operator does not impose any significant overhead. Once again, the implementation can be shown incrementally optimal if the underlying constraints support optimal algorithms. Blank entries in the table correspond to instances that could not be solved.

Table 3. Experimental results for the progressive party program

H/P	6	7	8	9	10
1–12,16	0.32 (0.30)	0.41 (0.35)	0.60 (0.54)	1.01 (0.69)	3.74 (1.27)
1–13	0.41 (0.34)	0.77 (0.46)	3.15 (0.99)	42.22 (5.80)	
1,3–13,19	0.41 (0.33)	0.79 (0.47)	3.50 (0.92)	28.6 (6.39)	
3–13,25,26	0.44 (0.35)	0.93 (0.50)	4.53 (1.30)	65.32 (7.79)	
1–11,19,21	1.75 (0.54)	36.1 (2.86)			
1–9,16–19	2.81 (1.06)	95.4 (4.81)			

2.5. *Partial Constraint Satisfaction*

Some local search algorithms do not rely on violation degrees; rather they reason on the truth values of the constraints only. This is the case, for instance, in partial constraint satisfaction [5], where the objective is to minimize the (possibly weighted) number of constraint violations. COMET provides an operator to transform an arbitrary constraint into a satisfaction constraint, i.e., a constraint whose violation degree is 0 or 1 only. For instance, the snippet

```
Constraint c = satisfactionConstraint(alldifferent(x));
```

assigns to *c* the satisfaction counterpart of the *alldifferent* constraint. The key contribution here is the systematic derivation of the satisfaction implementation in terms of the original constraint interface. The resulting implementation only induces a small constant overhead.

COMET also supports satisfaction systems that systematically apply the satisfaction operator to the constraints posted to them, simplifying the modeling and the declarative reading. Figure 5 illustrates satisfaction systems on an excerpt from a frequency allocation problem, where the objective is to minimize the number of violated constraints. Line 1 declares the satisfaction system *S*, while lines 5, 8, and 11 post the various types of distance constraints to the system.

3. Objective Functions

We now turn to objective functions, another class of differentiable objects in COMET. Objective functions may be linear, nonlinear, or may capture combinatorial substructures arising in many applications. A typical example is the function $MinNbDistinct(x_1, \dots, x_n)$ which minimizes the number of distinct values in (x_1, \dots, x_n) and arises in graph coloring, frequency allocation, and other resource allocation problems.

```

1.  SatisfactionSystem S(m);
2.  forall(d in DistanceCtrs)
3.    switch (d.ty) {
4.      case 1:
5.        S.post(abs(freq[d.v1]-freq[d.v2]) == d.rhs);
6.        break;
7.      case 2:
8.        S.post(abs(freq[d.v1]-freq[d.v2]) > d.rhs);
9.        break;
10.     case 3:
11.       S.post(abs(freq[d.v1]-freq[d.v2]) < d.rhs);
12.       break;
13.    }
```

Figure 5. Partial constraint satisfaction in frequency allocation.

This section illustrates how objective functions, like constraints, can be combined naturally to build more complex objectives. Once again, this desirable functionality comes from the Objective interface depicted in Figure 6 and implemented by all objective functions. In particular, the interface gives access to their variables, to two variables, and the differentiable API. The first variable (available through method `value`) maintains the *value* of the function incrementally. The second variable (available through method `evaluation`) maintains the *evaluation* of the function which may, or may not, differ from its value. The evaluation is useful to guide the local search more precisely by distinguishing states which have the same objective value. Consider again the objective function *nbDistinct*. Two solutions may use the same number of values, yet one of them may be closer than the other to a solution with fewer values. To distinguish between these, the evaluation may favor solutions where some values are heavily used while others have few occurrences. For instance, such an evaluation is used for graph coloring in [9] and is shown in Table 4 that summarizes the objective functions discussed in this paper. The differentiable API returns the variation of the evaluation induced by assignments and swaps. The method `getEvaluation` also returns the contribution of a variable to the evaluation and is the counterpart of method `getViolations` for objective functions. Note that several objective functions can coexist in the same COMET statement and they can be combined in many ways as discussed below. This flexibility is orthogonal to the pairs (value,evaluation) of objection functions.

3.1. Arithmetic Operators

Objective functions can be combined using traditional arithmetic operators. For instance, the excerpt

```
Objective f = condSum(open, fixed)
               + minAssignment(open, transportationCost);
```

illustrates the addition of two objective functions capturing combinatorial substructures expressing the fixed and transportation costs in uncapacitated warehouse location. These

```
interface Objective {
    var{int}[] getVariables();
    var{int} value();
    var{int} evaluation();
    int getEvaluation(var{int} x);
    int getAssignDelta(var{int} x,int v);
    int getSwapDelta(var{int} x1,var{int} x2);
    int getAssignDelta(var{int}[] x,int[] v);
}
```

Figure 6. The objective interface in COMET (partial description).

Table 4. The semantic of some objective functions

Objective	Value	Evaluation
$\text{MinNbDistinct}(x[E])$	$\#\{x[e] \mid e \in E\}$	$-\sum_i \{e \in E \mid x[e] = i\} ^2$
$\text{MinNbDistinct}(x[E], w[E])$	$\#\{x[e] \mid e \in E\}$	$\sum_{e \in E} w[x[e]]$
$\text{MaxNbDistinct}(x[E], w[V])$	$\#\{x[e] \mid e \in E\}$	$\sum_{v \in V: \sim \text{occur}(v, x)} w[v]$
$\text{condSum}(b[E], c[E])$	$\sum_{e \in E: b[e]} c[e]$	$\sum_{e \in E: b[e]} c[e]$
$\text{minAssignment}(b[V], c[E, V])$	$\sum_{e \in E} \min_{v \in V: b[v]} c[e, v]$	$\sum_{e \in E} \min_{v \in V: b[v]} c[e, v]$
$\text{constraintsAsObjective}(c)$	$v(c)$	$v(c)$

functions are useful in a variety of other applications such as k-median and configuration problems and encapsulate efficient incremental algorithms [13].

3.2. Reification: Constraints as Objective Functions

Some of the most challenging applications in combinatorial optimization feature complex feasibility constraints together with a “global” objective function. Some algorithms approach these problems by relaxing some constraints and integrating them in the objective function. To model such local search algorithms and relax constraint naturally, COMET provides the generic operator *constraintAsObjective* that transforms a constraint into an objective function whose value and evaluation are the violation degree of the constraint. Moreover, traditional arithmetic operators transparently apply this operator to simplify the declarative reading of the model. The implementation of this combinator, which is also given in Table 4, is in fact trivial, since it only maps one interface in terms of the other.

3.3. Frequency Allocation

To illustrate objective functions, consider a frequency allocation problem where feasibility constraints impose some distance constraints on frequencies and where the objective function consists of minimizing the number of frequencies used in the solution. We present an elegant COMET program implementing the Guided Local Search (GLS) algorithm proposed in [28].⁴ The key idea underlying the GLS algorithm is to iterate a simple local search where feasibility constraints are integrated inside the objective function. After completion of each local search phase, the weights of the violated constraints are updated to guide the search toward feasible solutions. If the solution is feasible, the weights of some frequencies used in the solutions (e.g., the values that occur the least) are increased to guide the search toward solutions with fewer frequencies.

Figure 7 depicts the modeling part of the GLS algorithm. Line 2 declares the decision variables, while lines 3 and 4 create the variables representing the weights that are

```

1.  LocalSolver m();
2.  var{int} freq[RV](m);
3.  var{int} w[RC](m) := 0;
4.  var{int} fw[RF](m) := 0;
5.  SatisfactionSystem S(m);

6.  forall(d in DistanceCtrs)
7.    switch (d.ty) {
8.      case 1:
9.        S.post(abs(freq[d.v1]-freq[d.v2]) == d.rhs,w[d.id]);
10.       break;
11.     case 2:
12.       S.post(abs(freq[d.v1]-freq[d.v2]) > d.rhs,w[d.id]);
13.       break;
14.     case 3:
15.       S.post(abs(freq[d.v1]-freq[d.v2]) < d.rhs,w[d.id]);
16.       break;
17.    }

18.  MinNbDistinct nbFreq(freq,fw);
19.  Objective obj = S + nbFreq;
20.  m.close();

```

Figure 7. The modeling part of a COMET program for frequency allocation.

associated with constraints and values respectively. Line 5 declares the satisfaction system S and lines 6–17 post the distance constraints in S , as presented earlier in the paper. The only difference is the use of weights which are useful to focus the search on violated constraints. Line 18 declares the objective function `nbFreq` which minimizes the number of distinct frequencies. Note that this function receives, as input, a dynamic weight for each value to guide the search toward solutions with few frequencies. Line 19 is particularly interesting: it defines the GLS objective function `obj` as the sum of the satisfaction system (a constraint viewed as an objective function) and the “actual” objective `nbFreq`. Of course, the search component uses the objective `obj`.

Figure 8 depicts the search part of the algorithm. Function `GLS` iterates two steps for a number of iterations: a local search, depicted in lines 8–19, and the weight adjustment. The weight adjustment is not difficult: it simply increases the weights of violated constraints and, if the solution is feasible, the weights of the frequencies that are used the least in the solution (modulo a normalization factor [28]). The local search considers all variables in a round-robin fashion and applies function `moveBest` on each of them, until a round does not improve the objective function (lines 8–14). Function `moveBest` selects, for variable `freq[v]`, the frequency `f` that minimizes the values of the objective function (ties are broken randomly). It uses the differentiable API to evaluate moves quickly.

It is particularly interesting to observe the simplicity and elegance of the COMET program. The modeling component simply specifies the constraints and the objective function, and combines them to obtain the GLS objective. The search component is expressed at a high level of abstraction as well, only relying on the objective function

```

1. function void GLS() {
2.   while (it < maxIterations) {
3.     localSearch();
4.     updateWeights();
5.     it++;
6.   }
7. }

8. function void localSearch() {
9.   int old;
10.  do {
11.    old = obj.evaluation();
12.    forall(v in RV) moveBest(v);
13.  } while (old != obj.evaluation());
14. }

15. function void moveBest(int v) {
16.   if (obj.getEvaluation(freq[v]) > 0)
17.     selectMin(f in Domain[v])(obj.getAssignDelta(freq[v],f))
18.     freq[v] := f;
19. }

```

Figure 8. The search part of a COMET program for frequency allocation.

and the decision variables. Table 5 depicts the experimental results on the first three instances of the Celar benchmarks. It reports quality and efficiency results for 50 runs of the algorithms on a 2.4 GHz Pentium IV. In particular, it gives the average, standard deviation, and the best and worst values for the number of frequencies and the time to the best solutions in seconds. It also reports the average number of iterations and its standard deviation. The preliminary results indicate that the resulting COMET program, despite its simplicity and elegance, compares well in quality and efficiency with specialized implementations in [28] whose quality results (best/worst/average) are reproduced in the last three columns of the table.

4. First-Class Expressions

We now turn to first-class expressions, another significant abstraction which is also an integral part of constraint programming libraries [8]. First-class expressions are

Table 5. Experimental results for frequency allocation

id	$\mu(S)$	$\sigma(S)$	B(S)	W(S)	$\mu(O)$	$\sigma(O)$	$\mu(I)$	$\sigma(I)$	B/W/ μ
1	18.76	2.15	16	24	4.40	2.50	778.60	420.02	16/22/18.6
2	14.00	0.00	14	14	0.69	1.06	152.72	257.28	14/14/14.0
3	15.36	1.23	14	18	2.69	1.96	523.40	383.69	14/18/15.4

constructed from incremental variables, constants, and arithmetic, logical, and relational operators. In COMET, first-class expressions are differentiable objects which can be evaluated to determine the effect of assignments and swaps on their values. In fact, several examples presented earlier feature first-class expressions. For instance, the COMET code

```
S.post(alldifferent(all(i in Size) queen[i] + i));
```

from the n -queens problem can be viewed as a shortcut for

```
expr{int} d[i in Size] = queen[i] + i;
S.post(alldifferent(d));
```

The first instruction declares an array of first-class integer expressions, element $d[i]$ being the expression $queen[i] + i$. The second instruction states the *alldifferent* constraint on the expression array.

First-class expressions significantly increase the modeling power of the language, since constraints and objective functions can now be defined over complex expressions instead of incremental variables only. Moreover, efficient implementations of these enhanced versions can be obtained systematically by combining the differentiable APIs of constraints, objective functions, and first-class expressions.

The all-interval series problem [1] further illustrates the benefits of using first-class expressions in COMET. The problem is a well-known exercise in music composition. It consists of finding a sequence of notes such that all notes in the sequence, as well as tonal intervals between consecutive notes, are different. It can thus be modeled as the finding of a permutation of the first n integers such that the absolute difference between two consecutive pairs of numbers are all different. To our knowledge, constraint programming can only find two structured solutions for instances of this problem, while musicians typically look for a variety of “innovative” solutions.

Figure 9 depicts a COMET program solving the all-interval series problem. The basic idea behind the modeling is to maximize the number of different distances

$$|v_2 - v_1|, \dots, |v_n - v_{n-1}|$$

between successive number v_i and v_{i+1} in the series. Line 7 in Figure 9 declares the variables $v[i]$ in the series, which are initialized to a random permutation of $0..n-1$. This guarantees that all variables have distinct values, a property maintained by the search procedure whose local moves swap the values of two variables. Line 8 specifies the objective function *MaxNbDistinct* which maximizes the number of distinct distances in $abs(v[2] - v[1]), \dots, abs(v[n] - v[n-1])$. Observe that almost all variables occur in two expressions.

It is interesting to observe the work performed by the implementation of *MaxNbDistinct* for this problem. The constraint first defines a set of variables e_i to maintain the expressions

$$e_i = |v_i - v_{i-1}|$$

```

1.  int n = 40;
2.  range Size = 1..n;
3.  range Domain = 0..n-1;
4.  range SD = 1..n-1;
5.  LocalSolver m();
6.  RandomPermutation perm(Domain);
7.  var{int} v[Size](m,Domain) := perm.get();

8.  MaxNbDistinct obj(all(k in SD) abs(v[k+1]-v[k]));
9.  m.close();

10. while (obj.value() < n-1)
11.   select(i in Size: obj.getEvaluation(v[i]) > 0)
12.     selectMax(j in Size:j!=i)(obj.getSwapDelta(v[i],v[j]))
13.       v[i] := v[j];

```

Figure 9. A COMET program for the all-interval series problem.

incrementally. It then states the objective $MaxNbDistinct(e_2, \dots, e_n)$ on these new variables. To evaluate the effect of swapping two variables v_i and v_j , the implementation performs the following two steps.

1. It retrieves the expression e_1, \dots, e_k whose values have changed to d_1, \dots, d_k due to the swap of v_i and v_j . There are typically four such expressions: e_i, e_{i+1}, e_j , and e_{j+1} .
2. It applies the differentiable method

$$\Delta_\alpha([e_1 \leftarrow d_1, \dots, e_k \leftarrow d_k], MaxNbDistinct(e))$$

The main benefit of first-order expressions is thus to automate this tedious and error-prone aspect of the local search, enabling the problem to be stated elegantly and to be solved efficiently.

The performance of the algorithm can be improved upon by associating weights to the distances (as suggested in [1]). The justification here is that larger distances are much more difficult to obtain and it is beneficial to bias the search toward them. To accommodate this enhancement, it suffices to replace line 8 by

```
MaxNbDistinct obj(all(k in SD)abs(v[k + 1] - v[k]), all(k in SD)k^3);
```

Table 6 gives the experimental results for various values of n and for the COMET program, extended to restart the computation after 1,000 iterations if no solution was found. The resulting COMET significantly outperforms existing programs. The Java implementation from [1] takes an average time of 63 seconds for $n = 30$ on a 733 MHz Pentium III (instead of 0.39 seconds for COMET on a 2.4 GHz machine). The gain probably comes from the better incrementality of COMET which uses differentiation to evaluate moves quickly. Indeed, our first COMET program for this problem, which did not use first-class expressions and did not support the differentiable API for the objective function, took 10.44 seconds in average over 50 runs for $n = 30$.

Table 6. Performance results on the all-interval series program

n	$\mu(T)$	$B(T)$	$W(T)$	$\sigma(T)$	$\mu(I)$	$\sigma(I)$
10	0.00	0.00	0.01	0.00	25.2	38.3
15	0.01	0.00	0.06	0.02	203.9	276.2
20	0.04	0.00	0.26	0.07	859.0	1235.8
25	0.07	0.00	0.52	0.10	1183.30	1419.4
30	0.39	0.01	2.41	0.64	6092.8	7986.0
35	0.86	0.03	6.75	1.49	11864.5	16792.8
40	2.62	0.09	18.68	4.05	32669.5	38641.3
45	6.63	0.12	36.79	9.24	78961.7	75349.3
50	34.09	1.27	165.41	52.56	355816.6	416601.4
55	130.05	9.30	278.75	160.56	1277633.4	906189.4

5. Generic Search Procedures

The combinators presented in this paper have an additional benefit: they provide the foundation for writing generic search procedures in COMET. Indeed, search procedures are now able to interact with declarative components only through the `Constraint` and `Objective` interfaces, abstracting away the actual details of constraints and objective functions. These generic search procedures do not depend on the application at hand, yet they exploit the differentiable APIs to implement heuristics and meta-heuristics efficiently. Moreover, these APIs are implemented by efficient incremental algorithms exploiting the structure of the applications. In other words, although the search procedures are generic and do not refer to specificities of the applications, they exploit their underlying structure through the combinators.

Figure 10 depicts a min-conflict search in COMET. The code is essentially similar to the search procedure in the queens problem: it is only necessary to collect the variable array and its range in lines 2–3, and to use the variable domains in line 6. As a consequence, lines 10–13 in the queens problem can simply be replaced by a call `minConflictSearch(S)`. Similarly, Figure 11 implements a constraint-directed search inspired by search procedures used in WSAT [21, 29] and DRAGONBREATH [16]. The key idea is to select a violated constraint first (line 6) and then a variable to re-assign (line 9). Once

```

1.  function void minConflictSearch(Constraint S) {
2.      var{int}[] var = S.getVariables();
3.      range Size = var.getRange();
4.      while (S.violations() > 0)
5.          selectMax(i in Size)(S.getViolations(var[i]))
6.              selectMin(v in var[i].getDomain())
7.                  (S.getAssignDelta(var[i],v))
8.                  var[i] := v;
9.      }

```

Figure 10. A generic min-conflict search in COMET.

```

1.  function int cdSearch(ConstraintSystem S) {
2.      UniformDistribution noise(0..99);
3.      range C = S.getRange();
4.      Constraint c[i in C] = S.getConstraint(i);
5.      while (S.violations() > 0)
6.          select(i in C: !c[i].isTrue()) {
7.              var{int}[] var = c[i].getVariables();
8.              range RV = var.getRange();
9.              selectMax(v in RV) (c[i].getViolations(var[v]))
10.             selectMin(val in var[v].getDomain())
                (c[i].getAssignDelta(var[v],val))
11.             if (S.getAssignDelta(var[v],val) < 0)
12.                 var[v] := val;
13.             else if (noise.get() < 10) {
14.                 var[v] := val;
15.             }
16.         }

```

Figure 11. A generic constraint-directed search in COMET.

again, lines 10–13 in the queens problem can simply be replaced by a call `cdSearch(S)`. Such constraint-oriented local search procedures are particularly effective on a variety of problems, such as the ACC sport-scheduling problem. The actual search procedure in [29] was also successfully implemented in COMET.

Observe that these search procedures are generic and do not depend on the actual shape of the constraints, which can be propositional, linear, nonlinear, combinatorial, or any combination of these. Moreover, generic search procedures bring another interesting benefit of constraint programming to local search: the ability to provide a variety of (parameterized) default search procedures, while exploiting the specific structure of the application.

6. Modeling and Reformulation

The main contribution of this paper is to show that the rich constraint language of constraint programming is also a natural and effective modeling vehicle for local search. It is thus legitimate to wonder whether traditional modeling and reformulation techniques from constraint programming are also beneficial for local search. This topic is beyond the scope of this paper, but it is interesting to highlight briefly the distinct roles of these techniques in constraint programming and local search.

6.1. Combinatorial Constraints

Combinatorial (global) constraints are fundamental in constraint programming to obtain more effective pruning algorithms. Their role in local search is different: they aim at exploiting combinatorial substructures for obtaining more efficient incremental algo-

rithms that maintain the violations and implement differentiation. Interestingly, the same combinatorial substructures are often appropriate for both purposes.

6.2. Redundant Constraints

Redundant constraints are a traditional modeling technique in constraint programming, whose role is also to improve the pruning of constraint propagation algorithms. The magic series problem illustrated the benefits of redundant constraints in local search. Their role in local search however is fundamentally different: they are used to refine the definition of violations in order to guide the search more effectively.

Figures 12 and 13 illustrate the benefits of redundant constraints on the magic series application. They depict the effect of assigning different values to the selected variable x on the violations of the redundant and cardinality constraints at two different iterations during the search. More precisely, each figure plots three functions:

1. the function $f_r(d) = \Delta_\alpha(x \leftarrow d, c_r)$ for the redundant constraint c_r , the selected variable x , and a given assignment α ;
2. the function $f_c(d) = \Delta_\alpha(x \leftarrow d, \{c_1, \dots, c_n\})$ for the cardinality constraints (c_1, \dots, c_n) , the selected variable x , and a given assignment α ;
3. the function $f_a(d) = \Delta_\alpha(x \leftarrow d, \{c_r, c_1, \dots, c_n\})$ for all constraints, the selected variable x , and a given assignment α .

First, observe that the redundant constraint provides much more differentiation between the various values. On Figure 12, the violations of cardinality constraints are almost similar for all possible assignments, while there are considerable variations in the violations of the redundant constraint. This is also the case for smaller values in Figure 13. Second, and most importantly, the redundant constraint actually recommends some values: the value $f_r(d)$ is negative for a number of values, while this is not

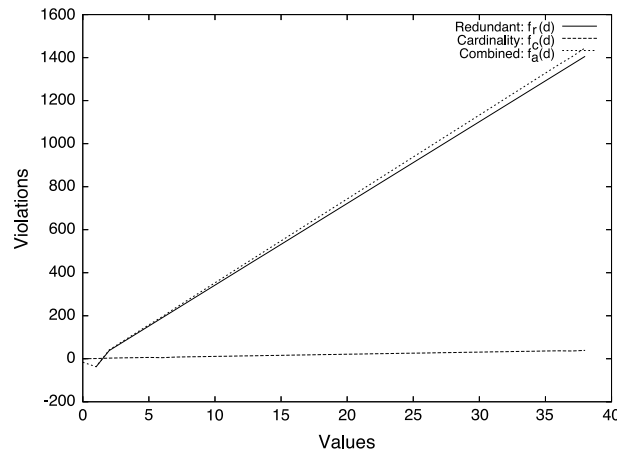


Figure 12. The role of redundant modeling: second illustration.

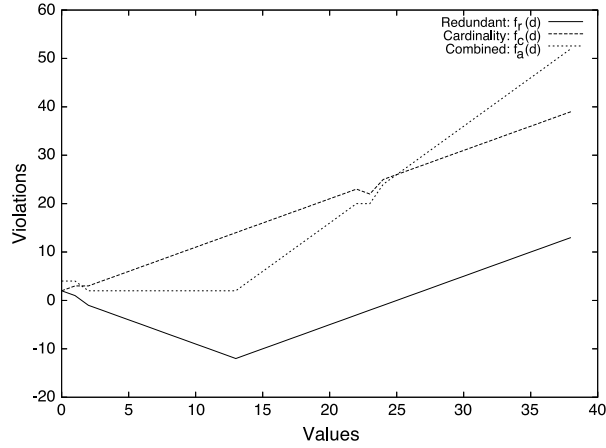


Figure 13. The role of redundant modeling: first illustration.

the case for the cardinality constraints. In particular, on Figure 12, the redundant constraint recommends the value 1, while the cardinality constraints have $f_c(d) \geq 0$ (and $f_c(d) = 0$ for small values). In fact, on this problem, the cardinality constraints almost always recommend the value 0 and this is the case for the two figures. Fortunately, the combination of the cardinality and redundant constraints guide the search much more precisely. For instance, on a magic series of length 10, the average number of iterations over 50 runs to reach a solution drops from 556 to 129.

Note also that constraint weights, as in the frequency allocation problem, and value weights, as in the all-interval series, are naturally viewed as redundant constraints, providing further evidence of the benefits of this modeling technique for local search.

6.3. Variable Elimination

Variable elimination is also an traditional reformulation technique in constraint programming (e.g., [7]) and its role is to improve the pruning of constraint propagation algorithms. Variable elimination was also used in the frequency allocation problem and, once again, its role in local search is of a different nature: it allows the local search to maintain the feasibility of the distance constraints $|f_i - f_j| = d$. Alternatively, it can be viewed as exploring a more complex neighborhood where assignments take place in pairs to maintain the feasibility of these specific distance constraints. Variable elimination is significant in that application, since otherwise the violations of these constraints increase each time one of their variables is assigned.

6.4. Symmetries

Symmetries have become a fundamental topic in constraint programming and it is interesting to discuss its role in local search as well. Some symmetry-breaking techniques,

such as the aggregation of undistinguishable values, are useful in local search, as shown in car sequencing [12] for instance. However, on other problems where symmetries seem fundamental in obtaining good efficiency for constraint programming, local search algorithms seem to behave well without taking symmetries into account. The reason is probably due to the nature of local search which typically explores the search space around the current solution and do not jump wildly to symmetric assignments in other regions of the search space. This is the case, for instance, of the social golfer problem [4]. Some authors have argued that some symmetry-breaking techniques, e.g., the use of lexicographic constraints, may actually hurt the performance of local search algorithms [18, 19]. Such constraints, which are not redundant, biased the violations in a negative way and may prevent the local search from moving closer to solutions. Note however that lexicographic constraints may also have a negative impact in constraint programming, where they may interfere with the search heuristic as discussed in [23], where it is argued that it is more effective to break symmetries during the search than to use lexicographic constraints.

7. Conclusion

This paper aimed at demonstrating that constraint-based combinators from constraint programming are natural abstractions for expressing local search algorithms. In particular, it showed that logical and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects encapsulating efficient incremental algorithms. These combinators, and their counterparts for objective functions, provide high-level ways of expressing complex ad-hoc constraints, generic search procedures, and partial constraint satisfaction. They were also shown to be amenable to efficient implementations. As a consequence, this paper, together with earlier results, indicates that the rich language of constraint programming is a natural vehicle for writing a wide variety of modular, extensible, and efficient local search programs.

Acknowledgments

Thanks to the reviewers for many interesting comments and suggestions. Pascal Van Hentenryck's research is partially supported by NSF ITR Awards DMI-0121495 and ACI-0121497.

Notes

1. COMET does not impose any limitations on how to generate initial solutions. For instance, they may be produced by greedy algorithms as is typically the case in scheduling (e.g., [2]).

2. The same issue arises in constraint programming of course. It may be possible to derive more effective pruning algorithm whenever additional structure is available.
3. Note that the labels of the first two rows (1–12,16 and 1–13) of Table 3 in [12] were switched.
4. The point is not to present the most efficient algorithm for this application, but to illustrate the concepts introduced herein on an interesting algorithm/application.

Reference

1. Codognet, C., & Diaz, D. (2001). Yet another local search method for constraint solving. In *Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications (SAGA 2001)*, pages 73–90. Berlin, Germany (December).
2. Dell’Amico, M., & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. *Ann. Oper. Res.* 41: 231–252.
3. Di Gaspero, L., & Schaerf, A. (2002). Writing local search algorithms using EASYLOCAL⁺⁺. In Voß, S. & Woodrubb, D.L., eds., *Optimization Software Class Libraries (Operations Research/Computer Science Interfaces Series)*, Kluwer, Boston, MA.
4. Dotú, I., Van Hentenryck, P., & Michel, L. (2005). Scheduling social golfers locally. In *Proceedings of the Second International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR’05)*. Prague.
5. Freuder, E. (1992). Partial constraint satisfaction. *Artif. Intell.* 58: 21–70.
6. Galinier, P., & Hao, J.-K. (2000). A general approach for constraint solving by local search. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR’00)*, pages 57–69. Paderborn, Germany (March).
7. Harvey, W., & Stuckey, P. (2003). Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2): 173–207.
8. Ilog Solver 4.4. (1998). Reference Manual. Ilog SA, Gentilly, France.
9. Johnson, D., Aragon, C., McGeoch, L., & Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Oper. Res.* 37(6): 865–893.
10. Laburthe, F., & Caseau, Y. (1998). SALSAs: A language for search algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP’98)*. Pisa (October).
11. Michel, L., & Van Hentenryck, P. (2000). Localizer. *Constraints*. 5: 41–82.
12. Michel, L., & Van Hentenryck, P. (2002). A constraint-based architecture for local search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 101–110. Seattle (November).
13. Michel, L., & Van Hentenryck, P. (2004). A simple Tabu search for warehouse location. *Eur. J. Oper. Res.* 157(3): 576–591.
14. Minton, S., Johnston, M. D., & Philips, A. B. (1990). Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 17–24. Boston (August).
15. Nareyek, A. (1998). *Constraint-Based Agents*. Springer Verlag, Berlin.
16. Nareyek, A. (2004). Dragonbreath. www.ai-center.com/projects/dragonbreath/.
17. Petit, T., Regin, J.-C., & Bessiere, C. (2001). Specific filtering Algorithms for over-constrained problems. In *Proceedings of 7th International Conference on the Principles and Practice of Constraint Programming (CP’01)*, pages 451–463. Paphos, Cyprus (November).
18. Prestwich, S. D. (2002). Super symmetric modeling for local search. In *Second International Workshop on Symmetry in Constraint Satisfaction Problems*.
19. Prestwich, S. D. (2003). Negative effects of modeling techniques on search performance. *Ann. Oper. Res.* 118: 137–150.

20. Ramalingam, G. (1993). *Bounded Incremental Computation*. Ph.D. Thesis. University of Wisconsin-Madison.
21. Selman, B., Kautz, H., & Cohen, B. (1994). Noise strategies for improving local search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343. Seattle (July).
22. Shaw, P., De Backer, B., & Furnon, V. (2002). Improved local Search for CP toolkits. *Ann. Oper. Res.* 115: 31–50.
23. Van Hentenryck, P. (2002). Constraint and integer programming in OPL. *INFORMS J. Comput.* 14(4): 345–372.
24. Van Hentenryck, P., & Deville, Y. (1991). The cardinality operator: A new logical connective and its application to constraint logic programming. In *The 8th International Conference on Logic Programming (ICLP-91)*, pages 745–759. Paris, France (June).
25. Van Hentenryck, P., & Michel, L. (2003). Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 65–80. Cork, Ireland.
26. Van Hove, W. J., Pesant, G., & Rousseau, L.-M. (2004). On global warming (Softening global constraints). In *Proceedings of 6th International Workshop on Preferences and Soft Constraints*, Toronto, Canada (September).
27. Voss, S., & Woodruff, D. (2002). *Optimization Software Class Libraries*. Kluwer Academic Publishers.
28. Voudouris, C., & Tsang, E. (1996). Partial constraint satisfaction problems and guided local search. In *Proceedings of the International Conference on Practical Applications of Constraint Technology (PACT-96)*, pages 337–356. London (April).
29. Walser, J. (1999). *Integer Optimization by Local Search*, Vol. 1637, Springer Verlag, GmbH, Heidelberg, Germany.