62. Integration of Metaheuristics and Constraint Programming

Luca Di Gaspero

A promising research line in the optimization community regards the hybridization of exact and heuristics methods. In this chapter we survey the specific integration of two complementary optimization paradigms, namely Constraint Programming, for the exact part, and metaheuristics.

62.1 Constraint Programming and Metaheuristics

Constraint programming (CP) [62.1, 2] is an effective methodology for the solution of combinatorial problems that has been successfully applied in many domains. In a nutshell, CP is a declarative programming paradigm based on the idea of describing the relations (i. e., constraints) between variables that must hold in all solutions of the combinatorial problem at hand. For example, in the solution to a Sudoku puzzle, the numbers to be placed must be unique with respect to columns, rows, and blocks of the board.

CP has an interdisciplinary nature, since it relies on contributions and methods from the communities of logic programming (LP), artificial intelligence (AI), and operations research (OR). Indeed, the simple declarative modeling language of CP, consisting of variables and constraints, is very similar to those available in classical LP languages such as Prolog. The solution method features constraint propagation which, in its essence, is a reasoning or inference procedure typical of AI. Finally, especially for optimization problems, the solution process makes use of OR inspired branch and bound procedures and/or of dedicated OR solvers for specific types of variables/constraints (e.g., the simplex method for real variables and linear constraints).

A CP model is an encoding of the problem statement using the basic CP building blocks, i. e., variables and constraints. Once a CP model of the problem under consideration has been stated, a CP solver is used to systematically search the solution space by alternating deterministic phases (constraint propagation) and nondeterministic phases (variable assignment, tree search), thus exploring implicitly or explicitly the whole search space. To this respect, CP belongs to the family of *complete* (or *exact*) solution methods. In other words, CP guarantees finding the (optimal) solution of the problem or proving that the problem is not satisfiable.

A different approach is usually taken by metaheuristics [62.3], such as local search [62.4], evolutionary algorithms [62.5], and ant colony optimization [62.6], just to name a few. These methods are *incomplete*, since they rely on heuristic information to focus on *interesting areas* of the search space and, in general, do not explore it entirely but are stopped after a given time limit. As a consequence, these algorithms do not guarantee finding the (optimal) solution, trading completeness for a (possibly) greater (empirical) efficiency in the solution process.

Just looking at completeness, it seems that the clear choice for solving combinatorial problems would be to always prefer CP over metaheuristics as the solution method. However, in practice completeness is hindered by the high computational effort due to the worst case complexity of the problems considered (usually NP-complete or NP-hard). Therefore, for practical purposes, also the execution of CP solvers is terminated before the whole search space has been explored and a number of heuristics is used to focus the search in the regions where it is more likely to find the solutions of the problem. Consequently, CP and metaheuristics could be seen as complementary approaches.

Although these two kinds of methods are have been individually studied by separated scientific communities (for historical reasons), in recent years we have witnessed an increasing interest in the integration of the methods. In many cases, indeed, each approach has its own strengths and weaknesses, and the general aim of method integration is to create hybrid algorithms that enhance the strengths of both approaches and (possibly) overcome some of the weaknesses. To this respect, *Yunes* maintains a web page listing a number of success stories of hybrid solution methods [62.7], that is, papers describing integrated approaches that outperform single optimization methods.

A number of conferences and workshops specifically aiming at bringing together researchers working on the integration of solution techniques for combinatorial problems have also recently started. Notable examples are the series of CP-AI-OR conferences [62.8, 9], started in 1999, and the Hybrid Metaheuristics workshops [62.10–16], started in 2004. The scope of these conferences is not limited to the integration of CP techniques with metaheuristics, but they also consider hybridization among other methods.

Additionally, a few surveys on the integration of complete methods with metaheuristics have appeared in the literature [62.17–19]. However, these surveys either deal with a particular class of metaheuristics (i. e., local search) [62.17, 19] and/or with a different class of complete methods (integer linear programming) [62.17, 18]. *Jourdan* et al. [62.20] also took CP methods into account, but they provide mostly a taxonomy of cooperation between optimization methods rather than surveying the specific integrations. *Wallace* and *Azevedo* et al. [62.21, 22] surveys hybrid algorithms, but from a constraint programming viewpoint and mainly in the settings of hybrid exact methods. In their recent review of hybrid metaheuristics *Blum* et al. [62.23] include a section on the integration of CP with local search and ant colony optimization (ACO). However, to the best of our knowledge, at present no specific survey on the integration of metaheuristics and constraint programming has been published in the literature. This work tries to overcome this lack and to review the different approaches specifically employed in the integration of CP methods within metaheuristics.

The chapter is organized as follows. In Sect. 62.2 the basic concepts of the constraint programming paradigm are introduced. They include modeling (Sect. 62.2.1), solution methods (Sect. 62.2.2), and CP systems (Sect. 62.2.3). The integration of CP with metaheuristics is presented in Sect. 62.3, which is organized on the basis of the metaheuristic type involved in the integration. Finally, in Sect. 62.4 some conclusions are drawn.

62.2 Constraint Programming Essentials

In this section, we will briefly describe the essential concepts of CP, which are needed to understand the following sections. The readers interested in a more detailed introduction to CP are referred to the book of *Apt* [62.1] and to the recent comprehensive *Handbook of Constraint Programming* [62.2].

In order to apply constraint programming to a combinatorial problem one first needs to model it through the specific formalism of constraint satisfaction or constrained optimization problems. Afterwards, the model can be solved by a CP solver, which alternates the analysis of constraints with tree search. Let us review these basic concepts.

62.2.1 Modeling

Constraint satisfaction problems (CSPs) are a useful formalism for modeling many real-world problems, either discrete or continuous. Remarkable examples are planning, scheduling, timetabling, just to name a few.

A CSP is generally defined as the problem of associating values (taken from a set of domains) to variables subject to a set of constraints. A solution of a CSP is an assignment of values to all the variables so that the constraints are satisfied. In some cases not all solutions are equally preferable and we can associate a cost function to the variable assignments. In these cases, we talk about constrained optimization problems (COPs), and we are looking for a solution that (without loss of generality) minimizes the cost value. These concepts are formally introduced in the following.

Constraint Satisfaction Problems

Given:

- $\mathcal{X} = \{x_1, \ldots, x_k\}$ is a set of *variables*.
- $\mathcal{D} = \{D_1, \ldots, D_k\}$ is a set of *domains* associated to the variables. In other words, each variable x_i can assume value d_i if and only if $d_i \in D_i$.
- $\mathcal{C}$ is a set of *constraints*, i. e., mathematical relations over $\mathbf{Dom} = D_1 \times \cdots \times D_k$.

We say that a tuple $\langle d_1, \ldots, d_k\rangle \in \mathbf{Dom}$ *satisfies* a constraint $C \in \mathcal{C}$ if and only if $\langle d_1, \ldots, d_k\rangle \in C$.

A *constraint satisfaction problem* (CSP) $\mathcal{P}$, described by the triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$, is the problem of finding the tuples $\bar{d} = \langle d_1, \ldots, d_k\rangle \in \mathbf{Dom}$ that satisfy every constraint $C \in \mathcal{C}$. Such tuples are called *solutions* of the CSP, and the set of solutions of $\mathcal{P}$ is denoted by $\mathbf{sol}(\mathcal{P})$.

$\mathcal{P}$ is said to be *consistent* or *satisfiable* if and only if $\mathbf{sol}(\mathcal{P}) \neq \emptyset$.

Notice that, depending on the modeling of the combinatorial problem at hand, we could be interested in determining different properties of the CSP. In the extreme case, for example, one could just want to know whether the problem is satisfiable, regardless of the actual solutions. The most common case is to search and provide a single solution to the problem, whereas sometimes one could be interested in all the solutions.

Constrained Optimization Problems

A *constrained optimization problem* (COP) $\mathcal{O} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, f\rangle$ is a CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$ with an associated *objective* function $f : \mathbf{sol}(\mathcal{P}) \to E$, where $\langle E, \preceq\rangle$ is a well-ordered set (typically, E is one of the sets $\mathbb{N}, \mathbb{Z}, \mathbb{R}$).

Differently from the previous case, the tuples $\bar{d} \in \mathbf{sol}(\mathcal{O})$ that satisfy every constraint are called *feasible solutions*, and the set of these tuples is usually assumed to be non-empty. A *solution* of the COP $\mathcal{O}$ is a feasible solution $\bar{e} \in \mathbf{sol}(\mathcal{O})$ for which the value of the objective function f is minimized, i. e.,

$$\forall \bar{d} \in \mathbf{sol}(\mathcal{O}) f(\bar{e}) \preceq f(\bar{d}) \, .$$

Observations

A few observations about this formalism are worth noting. First, notice that the general framework does not impose any restriction on either the type of domains and constraints or on the form of the objective function that can be used to express the problem. The basic type of domain is a finite set of integer values (also known as a *finite domain*), but there are other possibilities that enhance the expressive power of the modeling framework and capture some combinatorial substructures of the problem more naturally. For example, it is possible to deal with variables whose values are finite (multi)sets, (hyper)graphs, real valued intervals, or resources of a scheduling problem. Moreover, also the kind of constraints that can be employed is quite rich and includes arithmetic constraints, set constraints, permutation, counting and other types of combinatorial constraints, resource scheduling constraints, path constraints on graphs, and constraints expressible through regular expressions, just to name a few possibilities (see [62.24] for a comprehensive set of constraints and their implementation in actual CP systems).

These features clearly make the modeling phase easier and more precise with respect to other formalisms such as integer linear programming. Indeed, part of the combinatorial structure of the problem can be directly captured by the use of complex domains/constraints and, as for the objective function, there is no general limitation on its form, in particular, there is no assumption of linearity.

Another important point to be noticed regards the role of constraints. Differently from other modeling formalisms, which distinguish between constraints that *must* be satisfied (called *hard* constraints) and that *should preferably* be satisfied (*soft* constraints), in the original CSP/COP framework constraints are all hard and the solution methods, described in the following section, consider it mandatory to satisfy all of them. There have been several attempts in the CP literature to include soft constraints in the general framework (see, e.g., [62.25] for a review) but the most common way to handle them is to include a measure of their violation in the objective function of the problem.

62.2.2 Solution Methods

CP solution methods basically exploit a form of tree search that interleaves a branching phase with an analysis of constraints called constraint propagation. These two components are described in the following.

Branching and Tree Search

Once the combinatorial problem has been modeled as a CSP or a COP, CP solves it by constructing a solution by a process that exploits a non-deterministic *variable assignment*, where one value is selected together with one value in its current domain. This phase is also called *labeling* using (constraint) logic programming terminology, and a solution to the problem is a complete labeling. The process proceeds by recursively checking whether the current labeling can be extended to a consistent solution or, in the negative case, undoing the current assignment.

The pseudocode of the procedure, called (chronological) *backtracking*, is given in Algorithm 62.1. The procedure is at first called with the full set of variables and empty labeling as follows Backtracking($\mathcal{X}, \emptyset, C, \mathbf{Dom}$). The procedure performs an implicit form of tree search, where a *branch* is identified by the selection of one variable (a node of the search tree) and all the possible values for that variable (the edges).

Note that, at each step of the recursive procedure, the choice of the variable and the value to branch on is non-deterministic. Therefore, these choices are susceptible to heuristics to enhance performances.

In addition, there are also other possibilities to define a branching rule. For example, instead of selecting a possible value for the variable selected (i. e., the assignment $x_i := v$), the branching rule could split the domain of a given variable x_i in two by selecting a value $v \in D_i$ and adding the constraint $x_i \leq v$ on one branch and $x_i > v$ on the other.

Consistency and Constraint Propagation

The check for solution *consistency* does not need all the variables to be instantiated, in particular for detecting the unsatisfiability of the CSP with respect to some constraint. For example, in Algorithm 62.2, the most straightforward implementation of the procedure Consistent($L, C, \mathbf{Dom}$) is reported. The procedure simply checks whether the satisfiability of a given constraint can be ascertained according to the current labeling (i. e., if all of the constraint variables are assigned). However, the reasoning about the current labeling with respect to the constraints of the problem and the domains of the unlabeled variables does not necessarily need all the variables appearing in a constraint to be instantiated. Moreover, the analysis can prune (as a side effect) the domains of the unlabeled variables while preserving the set of solutions $\mathbf{sol}(\mathcal{P})$, making the exploration of the subtree more effective. This phase is called *constraint propagation* and is interleaved with the variable assignment. In general, the analysis of each constraint is repeated until a fixed point for the current situation is achieved. In the case that one of the domains becomes empty consistency cannot be achieved and, consequently, the procedure returns a fail.

Different notions of consistency can be employed. For example, one of the most common and most studied notions is hyper-arc consistency [62.26]. For a k-ary constraint C it checks the compatibility of a value v in the domain of one of the variables with the currently possible combinations of values for the remaining $k-1$ variables, pruning v from the domain if no supporting combination is found. The algorithms that maintain hyper-arc consistency have a complexity that is polynomial in the size of the problem (measured in terms of number of variables/constraints and size of domains). Other consistency notions have been introduced in the literature, each having different pruning capabilities and computational complexity, which are, usually, proportionally related to their effectiveness.

One of the major drawbacks of (practical) consistency notions is that they are *local* in nature, that is, they just look at the current situation (partial labeling and current domains). This means that it would be impossible to detect future inconsistencies due to the interaction of variables. A basic technique, called *forward checking*, can be used to mitigate this problem. This method exploits a one-step *look-ahead* with respect to the current assignment, i. e., it simulates the assignment of pair of variables, instead of a single one, thus evaluating the next level of the tree through a consistency notion. This technique can be generalized to several other problems.

Algorithm 62.1 Backtracking (U, L, C, Dom)

```
1:  if $U = \emptyset$ then
2:      return $L$
3:  end if
4:  pick variable $x_i \in U$
        /*possibly $x_i$ is selected non-deterministically*/
5:  for $v \in D_i$        /*Try to label $x_i$ with value $v$*/ do
6:      $\mathbf{Dom}' \leftarrow \mathbf{Dom}$
7:      if Consistent($L \cup \{x := v\}, C, \mathbf{Dom}'$)
         /*consistency notions can be different and have
        side effects on Dom*/ then
8:          $r \leftarrow$ Backtracking($U \setminus \{x\}, L \cup \{x := v\}, C, \mathbf{Dom}'$)
9:          if $r \neq$ fail then
10:             return  $r$ /*a consistent assignment has
                been found for the variables in $U \setminus \{x_i\}$ with
                respect to $x_i := v$*/
11:         end if
12:     end if
13: end for
```

```
14: return fail /*backtrack to the previous variable (no
    consistent assignment has been found for $x_i$)*/
```

Algorithm 62.2 Consistent (L, $\mathcal{C}$, **Dom**)

```
1: for $C \in \mathcal{C}$ do
2:   if all variables in $C$ are labeled in $L \wedge C$ is not satisfied by $L$ then
3:     return fail
4:   end if
5: end for
6: return true
```

Algorithm 62.3 BranchAndBound (U, L, $\mathcal{C}$, **Dom**, f, b, L_b)

```
1: if $U = \emptyset$ then
2:   if $f(L) < b$ then
3:     $b \leftarrow f(L)$  $L_b \leftarrow L$
4:   end if
5: else
6:   pick variable $x_i \in U$
     /*possibly $x_i$ is selected non-deterministically*/
7:   for $v \in D_i$ /*Try to label $x_i$ with value $v$*/ do
8:     $\mathbf{Dom}' \leftarrow \mathbf{Dom}$
9:     if Consistent($L \cup \{x := v\}, \mathcal{C}, \mathbf{Dom}'$) $\wedge$ bound($f, L \cup \{x := v\}, \mathbf{Dom}'$) $< b$
           /*additionally verify whether the current solution is bounded*/ then
10:      BranchAndBound($U \setminus \{x\}$, $L \cup \{x := v\}$, $\mathcal{C}$, $\mathbf{Dom}'$, $f$, $b$, $L_b$)
11:    end if
12:  end for
13: end if
```

Branch and Bound

In the case of a COP, the problem is solved by exploring the set **sol**($\mathcal{O}$) in the way above, storing the best value for f found as sketched in Algorithm 62.3. However, a constraint analysis (bound($f, L \cup \{x := v\}, \mathbf{Dom}'$)) based on a partial assignment and on the best value already computed, might allow to sensibly prune the search tree. This complete search heuristic is called (with a slight ambiguity with respect to the same concept in operations research) *branch and bound*.

62.2.3 Systems

A number of practical CP systems are available. They mostly differ with regards to the targeted programming language and modeling features available. For historical reasons, the first constraint programming systems were built around a Prolog system. For example, SICStus Prolog [62.27], was one of the first logic programming systems supporting constraint programming which is still developed and released under a commercial license. Another Prolog-based system specifically intended for constraint programming is ECL^iPS^e [62.28], which differently from SICStus Prolog is open source. Thanks to their longevity, both systems cover many of the modeling features described in the previous sections (such as different type of domains, rich sets of constraints, etc.).

Another notable commercial system specifically designed for constraint programming is the ILOG CP optimizer, now developed by IBM [62.29]. This system offers modeling capabilities either by means of a dedicated modeling language (called OPL [62.30]) or by means of a callable library accessible from different imperative programming languages such as C/C++, Java, and C#. The modeling capabilities of the system are mostly targeted to scheduling problems, featuring a very rich set of constructs for this kind of problems. Interestingly, this system is currently available at no cost for researchers through the IBM Academic Initiative.

Open source alternatives that can be interfaced with the most common programming languages are the C++ libraries of Gecode [62.31], and the Java libraries of Choco [62.32]. Both systems are well documented and constantly developed.

A different approach has been taken by other authors, who developed a number of modeling languages for constraint satisfaction and optimization problems that can be interfaced to different type of general purpose CP solvers. A notable example is MiniZinc [62.33], which is an expressive modeling language for CP. MiniZinc models are translated into a lower level language, called FlatZinc, that can be compiled and executed, for example, by Gecode, ECL^iPS^e or SICStus prolog.

Finally, a mixed approach has been taken by the developers of Comet [62.34]. Comet is a hybrid CP system featuring a specific programming/modeling language and a dedicated solver. The system has been designed with hybridization in mind and, among other features, it natively supports the integration of metaheuristics (especially in the family of local search methods) with CP.

62.3 Integration of Metaheuristics and CP

Differently from *Wallace* [62.21], we will review the integration of CP with metaheuristics from the perspective of metaheuristics, and we classify the approaches on the basis of the type of metaheuristic employed. Moreover, following the categorization of *Puchinger* and *Raidl* [62.18], we are mostly interested in reviewing the *integrative* combinations of metaheuristics and constraint programming, i. e., those in which constraint programming is embedded as a component of a metaheuristic to solve a subproblem or vice versa.

Indeed, the types of collaborative combinations are either straightforward (e.g., collaborative-sequential approaches using CP as a constructive algorithm for finding a feasible initial solution of a problem) or rather uninvestigated (e.g., parallel or intertwined hybrids of metaheuristics and CP).

62.3.1 Local Search and CP

Local search methods [62.4] are based on an iterative scheme in which the search moves from the current solution to an adjacent one on the basis of the exploration of a *neighborhood* obtained by perturbing the current solutions.

The hybridization of constraint programming with local search metaheuristics is the most studied one and there is an extensive literature on this subject.

CP Within Local Search

The integration of CP within local search methods is the most mature form of integration. It dates back to the mid 1990s [62.35], and two main streams are identifiable to this respect. The first one consists in defining the search of the candidate neighbor (e.g., the best one) as a constrained optimization problem. The neighborhoods induced by these definitions can be quite large, therefore, a variant of this technique is known by the name of *large neighborhood search* (LNS) [62.36]. The other kind of integration, lately named *constraint-based local search* (CBLS) [62.34], is based on the idea of expressing local search algorithms by exploiting constraint programming primitives in their control (e.g., for constraint checks during the exploration of the neighborhood) [62.37]. In fact, the two streams have a non-empty intersection, since the CP primitives employed in CBLS could be used to explore the neighborhood in a LNS fashion. In the following sections we review some of the work in these two areas.

A few surveys on the specific integration between local search and constraint programming exist, for example [62.38, 39].

Large Neighborhood Search. In LNS [62.36, 40] an existing solution is not modified just by applying small perturbations to solutions but a large part of the problem is perturbed and searched for improving solutions in a sort of *re-optimization* approach. This part can be represented by a set $\mathcal{F} \subseteq \mathcal{X}$ of *released* variables, called *fragment*, which determines the neighborhood relation $\mathcal{N}$. Precisely, given a solution $\bar{s} = \langle d_1, \ldots, d_k \rangle$ and a set $\mathcal{F} \subseteq \{X_1, \ldots, X_k\}$ of free variables, then

$$\mathcal{N}(s, \mathcal{F}) = \{\langle e_1, \ldots, e_k \rangle \in \mathbf{sol}(O) \\ : (X_i \notin \mathcal{F}) \to (e_i = d_i)\} .$$

Given $\mathcal{F}$, the neighborhood exploration is performed through CP methods (i. e., propagation and tree search).

The pseudocode of the general LNS procedure is shown in Algorithm 62.4. Notice that in the procedure there are a few hotspots that can be customized. Namely, one of the key issues of this technique consists in the criterion for the selection of the set $\mathcal{F}$ given the current solution $\bar{s}$, which is denoted by SelectFragment($\bar{s}$) in the algorithm. The most straightforward way to select it is to randomly release a percentage of the problem variables. However, the variables in $\mathcal{F}$ could also be chosen in a *structured* way, i. e., by releasing related variables simultaneously. In [62.41], the authors compare the effectiveness of these two alternative choices in the solution of a job-shop scheduling problem.

Also the upper bounds employed for the branch and bound procedure can be subject to a few design alternatives. A possibility, for example, is to set the bound value to $f(\bar{s}_b)$, the best solution value found that far, so that the procedure is forced to search at each step only for improving solutions. This alternative can enhance the technique when the propagation on the cost functions is particularly effective in pruning the domains of the released variables. At the opposite extreme, instead, the upper bound could be set to an infinite value so that a solution is searched regardless whether or not it is improving the cost function with respect to the current incumbent.

Moreover, another design point is the solution acceptance criterion, which is implemented by the AcceptSolution function. In general, all the classical local search solution acceptance criteria are applicable, obviously in dependence on the neighborhood selection criterion employed. For example, in the case of randomly released variables a Metropolis acceptance criterion could be adequate to implement a sort of simulated annealing.

Finally, the TerminateSearch criterion is one of those usually adopted in non-systematic search methods, such as the expiration of a time/iteration budget, either absolute or relative, or the discovery of an optimal solution.

*Algorithm 62.4 LargeNeighborhoodSearch (X, C, **Dom**, f)*

```
1:  create a (feasible) initial solution $\bar{s}_0 = \langle d_1^0, \dots, d_k^0 \rangle$
        /*possibly random or finding the first feasible
    solution of the full CP model*/
2:  $\bar{s}_b \leftarrow \bar{s}_0$
3:  $i \leftarrow 0$
4:  while not TerminateSearch($i, \bar{s}_i, f(\bar{s}_i), \bar{s}_b$) do
5:    $\mathcal{F} \leftarrow$ SelectFragment($\bar{s}_i$)
        /*strategy for selecting the released variables*/
6:    $L \leftarrow \{x_j := d_j^i : x_j \notin \mathcal{F}\}$
7:    $U \leftarrow \mathcal{F}$
8:    BranchAndBound($U, L, C, \mathbf{Dom}, f$,
      ChooseBounds($\bar{s}_i, \bar{s}_b$))
                          /*neighborhood exploration*/
9:    if AcceptSolution($L$) then
10:     $\bar{s}_{i+1} \leftarrow L$
11:     if $f(\bar{s}_{i+1}) < f(\bar{s}_b)$ then
12:       $\bar{s}_b \leftarrow \bar{s}_{i+1}$
13:     end if
14:   else
15:     $\bar{s}_{i+1} \leftarrow \bar{s}_i$
16:   end if
17:   $i \leftarrow i + 1$
18: end while
19: return $\bar{s}_b$
```

LNS has been successfully applied to routing problems [62.36, 42–45], nurse rostering [62.46], university course timetabling [62.47], protein structure prediction [62.48, 49], and car sequencing [62.50].

Cipriano et al. propose $\mathbb{GELATO}$, a modeling language and a hybrid solver specifically designed for LNS [62.51–53]. The system has been tested on a set of benchmark problems, such as the asymmetric traveling salesman problem, minimum energy broadcast, and university course timetabling.

The developments of the LNS technique in the wider perspective of very large neighborhood search (VLNS) was recently reviewed by *Pisinger* and *Ropke* [62.54]. *Charchrae* and *Beck* [62.55] also propose a methodological contribution to this area with some design principles for LNS.

Constraint-Based Local Search. The idea of encoding a local search algorithm by means of constraint programming primitives was originally due to *Pesant* and *Gendreau* [62.35, 56], although in their papers they focus on a framework that allows neighborhoods to be expressed by means of CP primitives. The basic idea is to extend the original CP model of the problem with a sort of surrogate model comprising a set of variables and constraints that intentionally describe a *neighborhood* of the current solution.

A pseudocode of CBLS defined along these lines is reported in Algorithm 62.5. The core of the procedure is at line 5, which determines the neighborhood model on the basis of the current solution. The main components of the neighborhood model are the new set of variables Y and constraints $C_{X,Y}$ that act as an interface of the neighborhood variables Y with respect to those of the original problem X. For example, the classical *swap* neighborhood, which perturbs the value of two variables of the problem by exchanging their values, can be modeled by the set $Y = \{y_1, y_2\}$, consisting of the variables to exchange, and with the interface constraints

$$(y_1 = i \wedge y_2 = j) \iff (x_i = s_j \wedge x_j = s_i)$$
$$\forall i, j \in \{1, \dots, n\} .$$

Moreover, an additional component of the neighborhood model is the evaluator of the move impact Δf, which can be usually computed incrementally on the basis of the single move.

It is worth noticing that the use of different modeling viewpoints is common practice in constraint programming. In classical CP modeling the different viewpoints usually offer a convenient way to express some constraint in a more concise or more efficient manner. The consistency between the viewpoints is maintained through the use of *channeling* constraints that link the different modelings. Similarly, although with a different purpose, in CBLS the linking between the full problem model and the neighborhood model is achieved through interface constraints.

Algorithm 62.5 ConstraintBasedLocalSearch (X, C_X, $\mathbf{Dom}_X$, f)

```
1:  create a (feasible) initial solution $\bar{s}_0 = \langle d_1^0, \dots, d_k^0 \rangle$
        /*possibly random or finding the first feasible
    solution of the original CP model*/
```

```
 2: $\bar{s}_b \leftarrow \bar{s}_0$
 3: $i \leftarrow 0$
 4: while not TerminateSearch$(i, \bar{s}_i, f(\bar{s}_i), \bar{s}_b)$ do
 5:    $\langle Y, C_{X,Y}, \mathbf{Dom}_Y, \Delta f\rangle \leftarrow$
       NeighborhoodModel$(\bar{s}_i)$
 6:    $L \leftarrow \emptyset$
 7:    $U \leftarrow Y$
 8:    BranchAndBound$(U, L, C_{X,Y}, \mathbf{Dom}_Y, \Delta f)$
       /*neighborhood exploration*/
 9:    if AcceptSolution$(L)$ then
10:       $\bar{s}_{i+1} \leftarrow$ Apply$(L, \bar{s}_i)$
11:       if $f(\bar{s}_{i+1}) < f(\bar{s}_b)$ then
12:          $\bar{s}_b \leftarrow \bar{s}_{i+1}$
13:       end if
14:    else
15:       $\bar{s}_{i+1} \leftarrow \bar{s}_i$
16:    end if
17:    $i \leftarrow i + 1$
18: end while
19: return $\bar{s}_b$
```

This stream of research has been revamped thanks to the design of the Comet language [62.34, 57], the aim of which is specifically to support declarative components inspired from CP primitives for expressing local search algorithms. An example of such primitives are *differentiable invariants* [62.58], which are declarative data structures that support incremental differentiation to effectively evaluate the effect of local moves (i. e., the Δf in Algorithm 62.5). Moreover, Comet support control abstractions [62.59, 60] specifically designed for local search such as the *neighbors* construct, which aims at expressing the unions of heterogeneous neighborhoods. Finally, Comet has been extended also to support distributed computing [62.61].

The embedding of local search within a constraint programming environment and the employment of a common programming language makes it possible to automatize the synthesis of CBLS algorithms from a high-level model expressed in Comet [62.62, 63]. The synthesizer analyzes the combinatorial structure of the problem, expressed through the variables and the constraints, and combines a set of basic *recommendations*, which are the basic constituents of the synthesized algorithm.

Other Integrations. The idea of exploring with local search a space of incomplete solutions (i. e., those where not all variables have been assigned a value) exploiting constraint propagation has been pursued, among others, by *Jussien* and *Lhomme* [62.64] for an open-shop scheduling problem. Constraint propagation employed in the spirit of forward checking and, more in general, look-ahead has been effectively employed, among others, by *Schaerf* [62.65] and *Prestwich* [62.66], respectively, for scheduling and graph coloring problems.

Local Search Within CP

Moving to the integration of local search within constraint programming, the most common utilization of local search-like techniques consists in limiting the exploration of the tree search only to paths that are "close" to a reference one. An example of such a procedure is *limited discrepancy search* (LDS) [62.67], an incomplete method for tree search in which only *neighboring* paths of the search tree are explored, where the proximity is defined in terms of different decision points called *discrepancies*. Only the paths (i. e., complete solutions) with at most k discrepancies are considered, as outlined in Algorithm 62.6.

*Algorithm 62.6 LimitedDiscrepancySearch (X, C, **Dom**, f, k)*

```
 1: $\bar{s}^* \leftarrow$ FirstSolution$(X, C, \mathbf{Dom}, f)$
 2: $\bar{s}_b \leftarrow \bar{s}^*$
 3: for $i \in \{1, \ldots, k\}$ do
 4:    for $\bar{t} \in \{\bar{s} : \bar{s}$ differs w.r.t. $\bar{s}^*$ for
                 exactly $i$ variables$\}$ do
 5:       if Consistent$(\bar{t}, \mathbf{Dom}) \wedge f(\bar{t}) < f(\bar{s}_b)$ then
 6:          $\bar{s}_b \leftarrow \bar{t}$
 7:       end if
 8:    end for
 9: end for
10: return $\bar{s}_b$
```

Another approach due to *Prestwich* [62.68] is called *incomplete dynamic backtracking*. Differently from LDS, in this approach proximity is defined among *partial* solutions, and when backtracking needs to take place it is executed by randomly unassigning (at most) b variables. This way, the method could be intended as a local search on partial solutions. In fact, the method also features other CP machinery, such as forward checking, which helps in boosting the search.

An alternative possibility is to employ local search in constraint propagation. Local probing [62.69, 70] is based on the partition of constraints into the set of *easy* and *hard* ones. At each choice point in the search tree the set of easy constraints is dealt with a local search metaheuristic (namely simulated annealing), while the hard constraints are considered by classical constraint propagation. This idea generalizes the approach of *Zhang* and *Zhang* [62.71], who first presented such a combination. Another similar approach was taken by *Sellmann* and *Harvey* [62.72], who used local search to propagate redundant constraints.

In [62.73] the authors discuss the incorporation of the tabu search machinery within CP tree search. In particular, they look at the memory mechanisms for limiting the size of the tree and the elite candidate list for keeping the most promising choices in order to be evaluated first.

62.3.2 Genetic Algorithms and CP

A genetic algorithm [62.5] is an iterative metaheuristic in which a population of strings, which represent candidate solutions, evolves toward better solutions in a process that mimics natural evolution. The main components of the evolution process are crossover and mutation operators, which, respectively, combine two parent solutions generating an offspring and mutate a given solution. Another important component is the strategy for the offspring selection, which determines the population at the next iteration of the process.

To the best of our knowledge, one of the first attempts to integrate constraint programming and genetic algorithms is due to *Barnier* and *Brisset* [62.74]. They employ the following genetic representation: given a CSP with variables $\{X_1,\dots,X_k\}$, the i-th gene in the chromosomes is related to the variable X_i and it stores a subset of the domain D_i that is allowed to be searched. Each chromosome is then decoded by CP, which searches for the best solution of the sub-CSP induced by the restrictions in the domains. The genetic operators used are a mutation operator that changes values on the subdomain of randomly chosen genes and a crossover operator that is based on a recombination of the set-union of the subdomains of each pair of genes. The method was applied to a vehicle routing problem and outperformed both a CP and a GA solver.

A different approach, somewhat similar to local probing, was used in [62.75] for tackling a production scheduling problem. In this case, the problem variables are split into two sets, defining two coupled subproblems. The first set of variables is dealt with by the genetic algorithm, which determines a partial schedule. This partial solution is then passed to CP for completing (and optimizing) the assignment of the remaining variables.

Finally, CP has been used as a post-processing phase for optimizing the current population in the spirit of memetic algorithms. In [62.76] CP actually acts as an unfeasibility repairing method for a university course timetabling problem, whereas in [62.77] the optimization performed by CP on a flow-shop scheduling problem is an alternative to the classical local search applied in memetic algorithms. This approach is illustrated in Algorithm 62.7.

Algorithm 62.7 A Memetic Algorithm with CP for Flow-Shop scheduling (adapted from [62.77])

```
1:  generate an initial population $P = \{p_1,\dots,p_l\}$ of
    permutations of $n$ jobs (each composed of $k$ tasks
    $\tau_{ij}$ whose start time and end time are denoted by $\sigma_{ij}$
    and $\eta_{ij}$ respectively)
2:  $g \leftarrow 0$
3:  **while** not TerminateSearch$(g, P, \min_{p\in P} f(p))$ **do**
4:      select $p_1$ and $p_2$ from $P$ by binary tournament
5:      $c \leftarrow p_1 \otimes p_2$ /*apply crossover*/
6:      **if** $f(c) \geq \min_{p\in P} f(p)$ **then**
7:          mutate $c$ under probability $p_m$
8:      **end if**
9:      decode $c = \langle c_1,\dots,c_n\rangle$ to the set of precedence
        constraints $C = \{\eta_{kc_j} \leq \sigma_{1c_{j+1}} : j=1,\dots,n-1\}$
10:     $L \leftarrow \emptyset$
11:     $U \leftarrow \{\sigma_{ij}, \eta_{ij} : i = 1,\dots,k, j = 1,\dots,n\}$
12:     BranchAndBound$(U, L, C \cup \{\eta_{ij} \leq \sigma_{i+1j} : i =$
        $1,\dots,k\}$, **Dom**$, f)$
13:     **if** $f(c) \geq \max_{p\in P} f(p)$ **then**
14:         discard $c$
15:     **else**
16:         select $r$ by reverse binary tournament
17:         $c$ replaces $r$ in $P$
18:     **end if**
19:     $g \leftarrow g + 1$
20: **end while**
21: **return** arg $\min_{p\in P} f(p)$
```

62.3.3 ACO and CP

Ant colony optimization [62.6] is an iterative constructive metaheuristic, inspired by ant foraging behavior. The ACO construction process is driven by a probabilistic model, based on pheromone trails, which are dynamically adjusted by a learning mechanism.

The first attempt to integrate ACO and CP is due to *Meyer* and *Ernst* [62.78], who apply the method for solving a job-shop scheduling problem. The proposed procedure employs ACO to learn the variable and value ordering used by CP for branching in the tree search. The solutions found by the CP procedure are fed back to the ACO in order to update its probabilistic model. In this approach, ACO can be conceived as a master online-learning branching heuristic aimed at enhancing the performance of a slave CP solver.

A slightly different approach was taken by *Khichane* et al. [62.79, 80]. Their hybrid algorithm works in two phases. At first, CP is employed to sample the space of feasible solutions, and the information collected is processed by the ACO procedure for updating the pheromone trails according to the solutions found by CP. In the second phase, the learned pheromone information is employed as the value ordering used for CP branching. This approach, differently from the previous one, uses the learning capabilities of ACO in an offline-learning fashion.

More standard approaches in which CP is used to keep track of the feasibility of the solution constructed by ACO and to reduce the domains through constraint propagation have been used by a number of authors. *Khichane* et al. apply this idea to job-shop scheduling [62.78] and car sequencing [62.79, 81]. Their general idea is outlined in Algorithm 62.8, where each ant maintains a partial assignment of values to variables. The choice to extend the partial assignment with a new variable/value pair is driven by the pheromone trails and the heuristic factors in lines 7–8 through a standard probabilistic selection rule. Propagation is employed at line 10 to prune the possible values for the variables not included in the current assignment.

Another work along this line is due to *Benedettini* et al. [62.82], who integrate a constraint propagation phase for Boolean constraints to boost a ACO approach for a bioinformatics problem (namely, haplotype inference). Finally, in the same spirit of the previous idea, *Crawford* et al. [62.83, 84] employ a look-ahead technique within ACO and apply the method to solve set covering and set partitioning problems.

Algorithm 62.8 Ant Constraint Programming (adapted from [62.79])

```
 1: initialize all pheromone trails to $\tau_{\max}$
 2: $g \leftarrow 0$
 3: **repeat**
 4:   **for** $k \in \{1, \dots, n\}$ **do**
 5:     $\mathcal{A}_k \leftarrow \emptyset$
 6:     **repeat**
 7:       select a variable $x_j \in X$ so that $x_j \notin \mathrm{var}(\mathcal{A}_k)$
          according to the pheromone trail $\tau_j$
 8:       choose a value $v \in D_j$ according to the
          pheromone trail $\tau_{jv}$ and a heuristic factor $\eta_{jv}$
 9:       add $\{x_j := v\}$ to $\mathcal{A}_k$
10:       Propagate$(\mathcal{A}_k, C)$
11:     **until** $\mathrm{var}(\mathcal{A}_k) = X$ **or** Failure
12:     update pheromone trails using $\{\mathcal{A}_1, \dots, \mathcal{A}_n\}$
13:   **end for**
14: **until** $\mathrm{var}(\mathcal{A}_i) = X$for some$i \in \{1, (\dots), n\}$ **or**
    TerminateSearch$(g, \mathcal{A}_i)$
```

62.4 Conclusions

In this chapter we have reviewed the basic concepts of constraint programming and its integration with metaheuristics. Our main contribution is the attempt to give a comprehensive overview of such integrations from the viewpoint of metaheuristics.

We believe that the reason why these integrations are very promising resides in the complementary merits of the two approaches. Indeed, on the one hand, metaheuristics are, in general, more suitable to deal with optimization problems, but their treatment of constraints can be very awkward, especially in the case of tightly constrained problems. On the other hand, constraint programming is specifically designed for finding feasible solutions, but it is not particularly effective for handling optimization. Consequently, a hybrid algorithm that uses CP for finding feasible solutions and metaheuristics to search among them has good chances to outperform its single components.

Despite the important steps made in this field during the last decade, there are still promising research opportunities, especially in order to investigate topics such as collaborative hybridization of CP and metaheuristics and validate existing integration approaches in the yet uninvestigated area of multiobjective optimization. We believe that further research should devote more attention to these aspects.

References

62.1 K.R. Apt: *Principles of Constraint Programming* (Cambridge Univ. Press, Cambridge 2003)

62.2 F. Rossi, P. van Beek, T. Walsh: *Handbook of Constraint Programming*, Foundations of Artificial Intelligence (Elsevier Science, Amsterdam 2006)

62.3 M. Dorigo, M. Birattari, T. Stützle: Metaheuristic. In: *Encyclopedia of Machine Learning*, ed. by C. Sammut, G.I. Webb (Springer, Berlin, Heidelberg 2010) p. 662

62.4 H.H. Hoos, T. Stützle: *Stochastic Local Search: Foundations & Applications* (Morgan Kaufmann, San Francisco 2004)

62.5 C. Sammut: Genetic and evolutionary algorithms. In: *Encyclopedia of Machine Learning*, ed. by C. Sammut, G.I. Webb (Springer, Berlin, Heidelberg 2010) pp. 456–457

62.6 M. Dorigo, M. Birattari: Ant colony optimization. In: *Encyclopedia of Machine Learning*, ed. by C. Sammut, G.I. Webb (Springer, Berlin, Heidelberg 2010) pp. 36–39

62.7 T. Yunes: Success stories in integrated optimization (2005) http://moya.bus.miami.edu/~tallys/integrated.php

62.8 W. J. van Hoeve: CPAIOR conference series (2010) available online from http://www.andrew.cmu.edu/user/vanhoeve/cpaior/

62.9 P. van Hentenryck, M. Milano (Eds.): *Hybrid Optimization: The Ten Years of CPAIOR*, Springer Optimization and Its Applications, Vol. 45 (Springer, Berlin 2011)

62.10 C. Blum, A. Roli, M. Sampels (Eds.): Hybrid Metaheuristics, First International Workshop (HM 2004), Valencia (2004)

62.11 M.J. Blesa, C. Blum, A. Roli, M. Sampels (Eds.): *Hybrid Metaheuristics: Second International Workshop (HM 2005)*, Lecture Notes in Computer Science, Vol. 3636 (Springer, Berlin, Heidelberg 2005)

62.12 F. Almeida, M.J. Blesa Aguilera, C. Blum, J.M. Moreno-Vega, M. Pérez, A. Roli, M. Sampels (Eds.): *Hybrid Metaheuristics: Third International Workshop*, Lecture Notes in Computer Science, Vol. 4030 (Springer, Berlin, Heidelberg 2006)

62.13 T. Bartz-Beielstein, M.J. Blesa Aguilera, C. Blum, B. Naujoks, A. Roli, G. Rudolph, M. Sampels (Eds.): *Hybrid Metaheuristics: 4th International Workshop (HM 2007)*, Lecture Notes in Computer Science, Vol. 4771 (Springer, Berlin, Heidelberg 2007)

62.14 M.J. Blesa, C. Blum, C. Cotta, A.J. Fernández, J.E. Gallardo, A. Roli, M. Sampels (Eds.): *Hybrid Metaheuristics: 5th International Workshop (HM 2008)*, Lecture Notes in Computer Science, Vol. 5296 (Springer, Berlin, Heidelberg 2008)

62.15 M.J. Blesa, C. Blum, L. Di Gaspero, A. Roli, M. Sampels, A. Schaerf (Eds.): *Hybrid Metaheuristics: 6th International Workshop (HM 2009)*, Lecture Notes in Computer Science, Vol. 5818 (Springer, Berlin, Heidelberg 2009)

62.16 M.J. Blesa, C. Blum, G.R. Raidl, A. Roli, M. Sampels (Eds.): *Hybrid Metaheuristics: 7th International Workshop (HM 2010)*, Lecture Notes in Computer Science, Vol. 6373 (Springer, Berlin, Heidelberg 2010)

62.17 I. Dumitrescu, T. Stützle: Combinations of local search and exact algorithms, Lect. Notes Comput. Sci. **2611**, 211–223 (2003)

62.18 J. Puchinger, G. Raidl: Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification, Lect. Notes Comput. Sci. **3562**, 113–124 (2005)

62.19 S. Fernandes, H. Ramalhinho Dias Lourenço: Hybrids combining local search heuristics with exact algorithms, V Congr. Esp. Metaheurísticas, Algoritm. Evol. Bioinspirados (MAEB2007), Tenerife, ed. by F. Rodriguez, B. Mélian, J.A. Moreno, J.M. Moreno (2007) pp. 269–274

62.20 L. Jourdan, M. Basseur, E.-G. Talbi: Hybridizing exact methods and metaheuristics: A taxonomy, Eur. J. Oper. Res. **199**(3), 620–629 (2009)

62.21 M. Wallace: Hybrid algorithms in constraint programming, Lect. Notes Comput. Sci. **4651**, 1–32 (2007)

62.22 F. Azevedo, P. Barahona, F. Fages, F. Rossi (Eds.): *Recent Advances in Constraints: 11th Annual ERCIM International Workshop on Constraint Solving and Contraint Logic Programming (CSCLP 2006)*, Lecture Notes in Computer Science, Vol. 4651 (Springer, Berlin, Heidelberg 2007)

62.23 C. Blum, J. Puchinger, G.R. Raidl, A. Roli: Hybrid metaheuristics in combinatorial optimization: A survey, Appl. Soft Comput. **11**(6), 4135–4151 (2011)

62.24 N. Beldiceanu, H. Simonis: Global constraint catalog (2011), available online from http://www.emn.fr/z-info/sdemasse/gccat/

62.25 P. Meseguer, F. Rossi, T. Schiex: Soft constraints. In: *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, ed. by F. Rossi, P. van Beek, T. Walsh (Elsevier, Amsterdam 2006)

62.26 A.K. Mackworth: Consistency in networks of relations, Artif. Intell. **8**(1), 99–118 (1977)

62.27 SICStus prolog homepage, available online from http://www.sics.se/isl/sicstuswww/site/index.html

62.28 K.R. Apt, M. Wallace: *Constraint Logic Programming Using Eclipse* (Cambridge Univ. Press, Cambridge 2007)

62.29 ILOG CP optimizer, available online from http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/

62.30 P. van Hentenryck: *The OPL Optimization Programming Language* (MIT Press, Cambridge 1999)

62.31 Gecode Team: Gecode: Generic constraint development environment (2006), available online from http://www.gecode.org
62.32 CHOCO Team: *Choco: An open source java constraint programming library*, Res. Rep. 10-02-INFO (Ecole des Mines de Nantes, Nantes 2010)
62.33 N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, G. Tack: Minizinc: Towards a standard CP modelling language, Lect. Notes Comput. Sci. **4741**, 529–543 (2007)
62.34 P.V. Hentenryck, L. Michel: *Constraint-Based Local Search* (MIT Press, Cambridge 2005)
62.35 G. Pesant, M. Gendreau: A view of local search in constraint programming, Lect. Notes Comput. Sci. **1118**, 353–366 (1996)
62.36 P. Shaw: Using constraint programming and local search methods to solve vehicle routing problems, Lect. Notes Comput. Sci. **1520**, 417–431 (1998)
62.37 B.D. Backer, V. Furnon, P. Shaw, P. Kilby, P. Prosser: Solving vehicle routing problems using constraint programming and metaheuristics, J. Heuristics **6**(4), 501–523 (2000)
62.38 F. Focacci, F. Laburthe, A. Lodi: Local search and constraint programming. In: *Handbook of Metaheuristics*, ed. by F. Glover, G. Kochenberger (Kluwer, Boston 2003) pp. 369–403
62.39 P. Shaw: Constraint programming and local search hybrids. In: *Hybrid Optimization*, Springer Optimization and Its Applications, Vol. 45, ed. by P. van Hentenryck, M. Milano (Springer, Berlin, Heidelberg 2011) pp. 271–303
62.40 L. Perron, P. Shaw, V. Furnon: Propagation guided large neighborhood search, Lect. Notes Comput. Sci. **3258**, 468–481 (2004)
62.41 E. Danna, L. Perron: Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs, Lect. Notes Comput. Sci. **2833**, 817–821 (2003)
62.42 Y. Caseau, F. Laburthe, G. Silverstein: A meta-heuristic factory for vehicle routing problems, Lect. Notes Comput. Sci. **1713**, 144–158 (1999)
62.43 L.M. Rousseau, M. Gendreau, G. Pesant: Using constraint-based operators to solve the vehicle routing problem with time windows, J. Heuristics **8**(1), 43–58 (2002)
62.44 S. Jain, P. van Hentenryck: Large neighborhood search for dial-a-ride problems, Lect. Notes Comput. Sci. **6876**, 400–413 (2011)
62.45 J.H.-M. Lee (Ed.): *Principles and Practice of Constraint Programming – CP 2011 – 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011, Proceedings*, Lecture Notes in Computer Science, Vol. 6876 (Springer, Berlin, Heidelberg 2011)
62.46 R. Cipriano, L. Di Gaspero, A. Dovier: Hybrid approaches for rostering: A case study in the integration of constraint programming and local search, Lect. Notes Comput. Sci. **4030**, 110–123 (2006)
62.47 H. Cambazard, E. Hebrard, B. O'Sullivan, A. Papadopoulos: Local search and constraint programming for the post enrolment-based course timetabling problem, Ann. Oper. Res. **194**(1), 111–135 (2012)
62.48 I. Dotu, M. Cebrián, P. van Hentenryck, P. Clote: Protein structure prediction with large neighborhood constraint programming search. In: *Principles and Practice of Constraint Programming*, ed. by I. Dotu, M. Cebrián, P. van Hentenryck, P. Clote (Springer, Berlin, Heidelberg 2008) pp. 82–96
62.49 R. Cipriano, A. Dal Palù, A. Dovier: A hybrid approach mixing local search and constraint programming applied to the protein structure prediction problem, Proc. Workshop Constraint Based Methods Bioinform. (WCB 2008), Paris (2008)
62.50 L. Perron, P. Shaw: Combining forces to solve the car sequencing problem, Lect. Notes Comput. Sci. **3011**, 225–239 (2004)
62.51 R. Cipriano, L. Di Gaspero, A. Dovier: A hybrid solver for Large Neighborhood Search: Mixing Gecode and EasyLocal++, Lect. Notes Comput. Sci. **5818**, 141–155 (2009)
62.52 R. Cipriano: On the hybridization of constraint programming and local search techniques: Models and software tools, Lect. Notes Comput. Sci. **5366**, 803–804 (2008)
62.53 R. Cipriano: On the Hybridization of Constraint Programming and Local Search Techniques: Models and Software Tools, Ph.D. Thesis (PhD School in Computer Science – University of Udine, Udine 2011)
62.54 D. Pisinger, S. Ropke: Large neighborhood search. In: *Handbook of Metaheuristics*, ed. by M. Gendreau, J.-Y. Potvin (Springer, Berlin, Heidelberg 2010) pp. 399–420, 2nd edn., Chap. 13
62.55 T. Carchrae, J.C. Beck: Principles for the design of large neighborhood search, J. Math. Model, Algorithms **8**(3), 245–270 (2009)
62.56 G. Pesant, M. Gendreau: A constraint programming framework for local search methods, J. Heuristics **5**(3), 255–279 (1999)
62.57 L. Michel, P. van Hentenryck: A constraint-based architecture for local search, Proc. 17th ACM SIGPLAN Object-oriented Program. Syst. Lang. Appl. (OOPSLA '02), New York (2002) pp. 83–100
62.58 P. van Hentenryck, L. Michel: Differentiable invariants, Lect. Notes Comput. Sci. **4204**, 604–619 (2006)
62.59 P. van Hentenryck, L. Michel: Control abstractions for local search, J. Constraints **10**(2), 137–157 (2005)
62.60 P. van Hentenryck, L. Michel: Nondeterministic control for hybrid search, Lect. Notes Comput. Sci. **3524**, 863–864 (2005)
62.61 L. Michel, A. See, P. van Hentenryck: Distributed constraint-based local search, Lect. Notes Comput. Sci. **4204**, 344–358 (2006)
62.62 P. van Hentenryck, L. Michel: Synthesis of constraint-based local search algorithms from

high-level models, 22nd Natl. Conf. Artif. Intell. AAAI, Vol. 1 (2007) pp. 273–278
62.63 S.A. Mohamed Elsayed, L. Michel: Synthesis of search algorithms from high-level cp models, Lect. Notes Comput. Sci. **6876**, 256–270 (2011)
62.64 N. Jussien, O. Lhomme: Local search with constraint propagation and conflict-based heuristic, Artif. Intell. **139**(1), 21–45 (2002)
62.65 A. Schaerf: Combining local search and look-ahead for scheduling and constraint satisfaction problems, 15th Int. Joint Conf. Artif. Intell. (IJCAI-97), Nagoya (1997) pp. 1254–1259
62.66 S. Prestwich: Coloration neighbourhood search with forward checking, Ann. Math. Artif. Intell. **34**, 327–340 (2002)
62.67 W.D. Harvey, M.L. Ginsberg: Limited discrepancy search, 14th Int. Joint Conf. Artif. Intell., Montreal (1995) pp. 607–613
62.68 S. Prestwich: Combining the scalability of local search with the pruning techniques of systematic search, Ann. Oper. Res. **115**(1), 51–72 (2002)
62.69 O. Kamarainen, H. Sakkout: Local probing applied to scheduling, Lect. Notes Comput. Sci. **2470**, 81–103 (2006)
62.70 O. Kamarainen, H. El Sakkout: Local probing applied to network routing, Lect. Notes Comput. Sci. **3011**, 173–189 (2004)
62.71 J. Zhang, H. Zhang: Combining local search and backtracking techniques for constraint satisfaction, Proc. 13th Natl. Conf. Artif. Intell. (AAAI96) (1996) pp. 369–374
62.72 M. Sellmann, W. Harvey: Heuristic constraint propagation, Lect. Notes Comput. Sci. **2470**, 319–325 (2006)
62.73 M. Dell'Amico, A. Lodi: On the integration of metaheuristic stratgies in constraint programming. In: *Metaheuristic Optimization Via Memory and Evolution: Tabu Search and Scatter Search*, Operations Research/Computer Science Interfaces, Vol. 30, ed. by C. Rego, B. Alidaee (Kluwer, Boston 2005) pp. 357–371, Chap. 16
62.74 N. Barnier, P. Brisset: Combine & conquer: Genetic algorithm and CP for optimization, Lect. Notes Comput. Sci. **1520**, 463–463 (1998)
62.75 H. Hu, W.-T. Chan: A hybrid GA-CP approach for production scheduling, 5th Int. Conf. Nat. Comput. (2009) pp. 86–91
62.76 S. Deris, S. Omatu, H. Ohta, P. Saad: Incorporating constraint propagation in genetic algorithm for university timetable planning, Eng. Appl. Artif. Intell. **12**(3), 241–253 (1999)
62.77 A. Jouglet, C. Oguz, M. Sevaux: Hybrid flowshop: a memetic algorithm using constraint-based scheduling for efficient search, J. Math. Model Algorithms **8**(3), 271–292 (2009)
62.78 B. Meyer, A. Ernst: Integrating ACO and constraint propagation, Lect. Notes Comput. Sci. **3172**, 166–177 (2004)
62.79 M. Khichane, P. Albert, C. Solnon: CP with ACO. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ed. by L. Perron, M.A. Trick (Springer, Berlin, Heidelberg 2008) pp. 328–332
62.80 M. Khichane, P. Albert, C. Solnon: Strong combination of ant colony optimization with constraint programming optimization, Lect. Notes Comput. Sci. **6140**, 232–245 (2010)
62.81 M. Khichane, P. Albert, C. Solnon: Integration of ACO in a constraint programming language, Lect. Notes Comput. Sci. **5217**, 84–95 (2008)
62.82 S. Benedettini, A. Roli, L. Di Gaspero: Two-level ACO for haplotype inference under pure parsimony, Lect. Notes Comput. Sci **5217**, 179–190 (2008)
62.83 B. Crawford, C. Castro: Integrating lookahead and post processing procedures with ACO for solving set partitioning and covering problems, Lect. Notes Comput. Sci. **4029**, 1082–1090 (2006)
62.84 B. Crawford, C. Castro, E. Monfroy: Constraint programming can help ants solving highly constrained combinatorial problems, ICSOFT 2008 – Proc. 3rd Int. Conf. Software Data Technol., INSTICC, Porto (2008) pp. 380–383