Constraint Programming and Local Search Hybrids

Paul Shaw

Abstract Constraint programming and local search are two different optimization paradigms which, over the last two decades or so, have been successfully combined to form hybrid optimization techniques. This chapter describes and compares a number of these works, with the goal of giving a clear picture of research in this domain. We close with some open topics for the future.

1 Introduction

This chapter describes ways in which constraint programming (CP) and local search (LS) can be usefully combined. Researchers have been looking at ways to combine these two approaches for around 20 years now. As we shall see, these combinations can take on a variety of guises: diverse examples include using LS in propagation and pruning rules, using CP to make moves in an LS process, and performing local moves on the decision path in a search tree. Focacci et al. describe a number of these in [28].

LS and CP offer two different and contrasting ways of solving combinatorial optimization problems. LS works on a complete assignment of values to variables and navigates through the search space by making "moves": each move is selected from one available in the "neighborhood" and modifies part of the complete assignment. Roughly speaking, the neighborhood is the set of possible moves available from an assignment (see [1] for elaboration) – we will not need a more formal definition here.

CP can be seen as both a modeling and a solving technology. In terms of modeling, CP provides high-level structural constraints, such as the element constraint and all-different, very often resulting in more compact and readable models than alternative approaches, such as integer programming. In terms of solving, CP

P. Shaw (✉)
IBM, 1681 route des Dolines, 06560 Valbonne, France
e-mail: paul.shaw@fr.ibm.com

builds up a partial solution constructively by heuristically making assignments to variables (which can later be undone through backtracking). Before each heuristic assignment, inference rules and algorithms are used to reduce the set of possible values for variables not yet in the partial solution (constraint propagation).

We examine different ways that CP and LS methods have been used together to solve combinatorial optimization problems. This chapter does not, however, explore what might be considered the more "classic" methods of combining complete and LS techniques which either: (a) Run an LS method to provide a good upper bound on the objective function, then switch to a complete method for establishing proof, or, (b) run an LS method on each solution found by a tree-based method, in order to produce better solutions faster and provide tighter bounds for the complete method, or, (c) more complex combinations of the above where the two methods are perhaps run in parallel with communication of solutions and bounds being carried out throughout the search process. These kinds of methods, although interesting in their own right, have no character specific to CP and are not mentioned further here.

2 Local Search on CP Models

In this chapter, when we refer to combinations of LS and CP, we are, for the most part, referring to a *combination of searches*: a fusion of an inherently LS method with the classic constructive incremental process of CP. In contrast to this general theme, this first section describes combinations using only the *modeling* aspects of CP in combination with LS methods for finding and improving solutions to these models.

2.1 Min-Conflicts and Related Algorithms

Min-conflicts is an appealing technique described in [64]. The idea is both simple and effective, and based on the authors' observations of the Guarded Discrete Stochastic (GDS) network [2], a system based on neural networks.

In min-conflicts, and indeed in many of the methods discussed in the chapter, constraints may be violated during the search process, and that violation is often measured and treated as an objective function in the classical sense.[1] The essential idea of min-conflicts is to always take a local decision which minimizes the total number of conflicts. One may consider the number of conflicts for a constraint as (an approximation of) the number of variables that must change value in order to

[1] These techniques (min-conflicts included) are often geared toward the solution of decision problems, but optimization problems can be solved in the usual manner, through a series of such decision problems with a tightening upper bound on the objective function.

satisfy it. For binary CSPs, the number of constraints violated is typically used. Two versions of min-conflicts are offered up in [64]: one based on LS (min-conflicts hill climbing), the other based on a complete tree-search technique. We concentrate only on the LS version here, but the other is mentioned in Sect. 5.

In min-conflicts, an initial solution to the problem is generated by assigning values to variables in turn, each instantiation choosing a value for the variable under investigation which minimizes the number of conflicts with variables that have already been given values. When this first "greedy" solution has been generated, min-conflicts enters a local improvement phase which tries to reduce the number of conflicts to zero by the following process:

```
while conflicts exist
    choose a variable $x$ in conflict
    choose a value for $x$ which minimizes conflicts on $x$
end while
```

How variables and values are chosen in case of ties (*what* conflicting variable, *which* value which minimizes conflicts), can depend upon implementation, but [64] suggests using random tie-breaking for the value.

To the best of our knowledge, the authors of the min-conflicts heuristic were the first to clearly communicate on a major reason that LS can work so much better than constructive search when good solutions are quickly sought. In [64], when speaking of the GDS network, the authors write:

> Our second hypothesis was that the network's search process uses information about the current assignment that is not available to a constructive backtracking program.

The point here is that the LS algorithm always has *global* information available to it on the quality of the current state of the algorithm: all variables contribute to its evaluation and the evaluation of its next move. By contrast, a constructive method has comparatively little information on which to base its next decision. Methods such as *promise* [34], *constrainedness* [37], and *impacts* [87] have been put forward for constructive backtracking algorithms, but their evaluations are essentially statistical in nature and hold none of the solidity of the conflicts measure.

Min-conflicts worked successfully, comparing so well to the GDS network that min-conflicts replaced it in the long-term scheduling component of the Hubble Space Telescope. However, it was realized by the authors and others that min-conflicts could fall foul of local minima. A simple example: consider the system, $A, B, C \in 0, 1$, $A + B + C = 1$, $A \neq C$, $B \neq C$. The only solution to this system is $ABC = 001$ (goal state). Further assume that the initial solution is 100, produced by assigning A,B, and C in order, minimizing conflicts on the way. This initial solution violates only one constraint: $B \neq C$, and so has a cost of 1. From this initial position, to reach the goal, the assignment must pass through one of the three neighboring assignments of the goal state: namely 101, 011, or 000 (recall that min-conflicts may only change one variable assignment at a time). Table 1 shows

Table 1 Cost contributions of assignments of min-conflicts at a local minimum

Assignment		Contribution			
Label	ABC	$A + B + C = 1$	$A \neq C$	$B \neq C$	Total
Start	100	0	0	1	1
Goal	001	0	0	0	0
Goal Nei. 1	000	1	1	1	3
Goal Nei. 2	011	1	0	1	2
Goal Nei. 3	101	1	1	0	2

the cost of the five important assignments (start state, goal state, and the three neighbors of the goal state). Notice that it is impossible for min-conflicts to reach the goal state from the start state as all states neighboring the goal are of higher cost than the starting state.

The local minimum problem of min-conflicts was addressed by the breakout method [66] and GENET [24].[2] The breakout method is largely inspired by min-conflicts, whereas GENET takes much of its design from GDS (which was the system on which min-conflicts itself was based). Both methods escape local minima by adjusting the way in which constraint violations are counted: each constraint is assigned a weight (initially one), which may be modified by a learning process.

In both methods, a greedy min-conflicts type of approach is pursued until a local minimum is reached. In the breakout method, a local minimum is one where the number of violated constraints cannot be strictly decreased. In GENET, sideways (cost neutral) moves are allowed and the greedy phase terminates when the network has had the same state for two successive iterations. When in the local minimum state, both methods increase the weights of *all* currently violated constraints by one, and the greedy process is restarted from the current point. In experiments, both methods are shown to strongly dominate the min-conflicts hill climbing method.

Yugami et al. [115] describe another method for breaking out of local minima, named EFLOP. The idea of EFLOP is that it can be used as a diversification method that a hill climber can call when confronted with a local minimum. EFLOP will change the value of a conflicting variable, even if that increases the number of conflicts, and then will go on to try to repair the new conflicts created, without ever changing the value of a variable more than once. In this way, the repair effect propagates through the network until no newly created conflicts can be repaired. The resulting assignment is then given back to the hill climber as a point from which to continue. The method is similar in spirit to that of an ejection chain in LS (see for example [113]) where changing a solution element has a cascading effect on others. The authors report significant gains when the EFLOP module is plugged into the min-conflicts hill climber or GSAT [98].

[2] Later, GENET was generalized to Guided Local Search [110], a meta-heuristic based on penalization of the cost function.

2.2 SAT-Based Methods

Another well-used technique for solving CSPs using LS methods is to translate the CSP problem to an SAT problem, and then use one of the well-known SAT solving techniques (such as WalkSAT [97]).

There are various ways of translating CSPs to SAT. In [112], Walsh describes two methods: the direct encoding and the logarithmic encoding. In the direct encoding, one propositional variable x_{ij} is introduced for each unary assignment $X_i = j$ in the CSP. Clauses of the form $\vee_{j \in \mathrm{Domain}(X_i)} x_{ij}$ are introduced to assure that each CSP variable is assigned a value. Finally, each constraint is encoded by introducing one clause for each tuple of incompatible values. For example, if $X_1 = 1$ is incompatible with $X_2 = 3$, then this translates to the clause $\neg x_{11} \vee \neg x_{23}$. Note that clauses assuring that each CSP variable must take on *exactly one value*, rather than at least one value are not necessary. If, for example, a solution is found with both x_{11} and x_{12} true, then either the value 1 or 2 can be chosen for the variable X_1: both will give legal solutions.

In the logarithmic encoding, $b_i = \lceil \log_2(|\mathrm{Domain}(X_i)|) \rceil$ variables are introduced to encode the binary representation of the value of X_i. This representation is more compact (in terms of variables) than the direct encoding, but the constraints of the CSP translate to longer clauses (proportional to the sums of the logs of the domain sizes of the variables in the constraint). Furthermore, for any i, if b_i is not a power of two, then additional clauses are required to rule out the combinations of values which do not correspond to a domain value of X_i in the CSP. Walsh showed that in both encodings, the unit propagation rule performs less filtering than arc consistency on the original problem.

Gent [38] describes a SAT encoding of CSPs called the *support encoding* which he showed as performing equivalent filtering to arc consistency on the original problem. The support encoding uses one propositional variable per CSP domain value as for the direct encoding but describes the constraints differently. The supports of each domain value are described by clauses. If, for example, on a binary constraint between X_1 and X_2, value 1 of X_1 is only compatible with values 3 and 5 of X_2, this would translate as a clause: $x_{11} \Rightarrow x_{23} \vee x_{25}$. In this formulation, clauses are also added to ensure that each CSP variable takes exactly one value. Of particular interest is that this formulation seems quite compatible with the operation of LS methods: a variant of WalkSAT performed on average over an order of magnitude faster on the support encoding than on the direct encoding.

More recently, more complete translations of CP models have been proposed: Huang [46] describes a universal framework for translating CP models to SAT models based on the logarithmic encoding, translating much of the MiniZinc language [102]. The Sugar solver [105, 106] also operates by translating CSP models to SAT, but in this case by using an *order encoding*, where each SAT variable $y_{x,a}$ represents the truth value of the relation $x \leq a$ in the original CSP formulation.

2.3 Localizer and Comet

Localizer [62, 63] was the first attempt at creating a language for the specification of LS algorithms. The modeling language is reminiscent of languages for CP, but with differences. First of all, the notion of an expression (say $x + y$) is central. Furthermore, when one writes $z = x + y$ in Localizer, this means that the value of z is *computed* from the values of x and y. (z is not a variable, has no domain, and its value cannot be decided by the system by a search procedure.) Localizer refers to the $\langle lhs \rangle = \langle \text{expr} \rangle$ construct as an *invariant*, and ensures that at any time during the computation, in our example of $z = x + y$, the value of z is equal to the value of x plus the value of y. This is performed by incremental calculations to maintain the value of z whenever x or y changes value. Invariants can also be created over more complex structures, such as sets. This means that Localizer can maintain, for example, the set of best variables to flip in a GSAT-type implementation, as shown in Fig. 1. The set *Candidates* maintains the set of variables which are among the best to flip. This set is updated incrementally whenever one of the *gain* expressions changes. These *gain* expressions are themselves computed from the current SAT assignment and a statement of the clauses of the problem (not shown). Localizer then makes a move by changing the value of one of these variables to its negation.

Localizer maintains invariants through a differencing approach, which we briefly describe. Consider the expressions and invariants in a Localizer program to form a directed acyclic graph. Each node represents an expression, with a directed arc connecting expression a to expression b when b directly mentions a.

Consider that arcs in the graph have the same rank as their source nodes. Nodes with no incoming arcs are labeled with rank 0. Other nodes are ranked one more than their highest ranked incoming arc. Consider the example $z = \sum_{1 \le i \le n} x_i + \sum_{2 \le i \le n} x_{i-1} x_i$. A graph corresponding to this invariant is shown in Fig. 2. In this graph, all "x" nodes have rank 0, all multiplication nodes have rank 1, and the node corresponding to z has rank 2.

Localizer proceeds by labeling each node according to its topological rank in its "planning" phase. Then, whenever the state of the system changes (here, expressed by the x_i expressions), the "execution" phase incrementally maintains the value of z by minimally recomputing parts of z following the topological order. Essentially, all changes are processed at level k before moving to level $k + 1$. This means that no part of an invariant is considered more than once, and node values are never changed

```
maxGain: int = max (i in 1..n) gain[i];
Candidates: {int} = {
  i : int | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0
};
...
move a[i] := !a[i]
where i from Candidates
```

Fig. 1 A Localizer code using a GSAT technique (taken from [63])

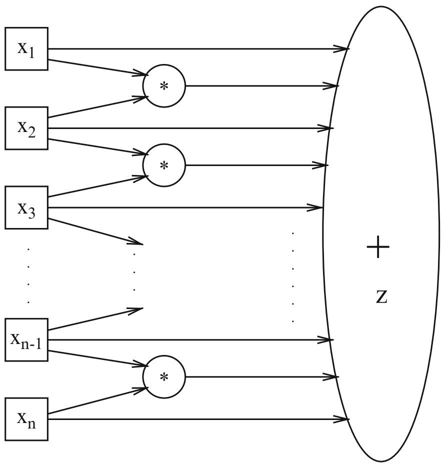

Fig. 2 Graph representation of a localizer invariant

more than once. The recalculation of the invariant is done in an incremental way. At level $k+1$, the current and previous values (before the update) of nodes at level k are available in order to recalculate the nodes at level $k+1$. For example, if the value of x_2 changes, then x_1x_2 and x_2x_3 are recalculated (in constant time), then z is recalculated in constant time by applying the *difference of the new and old values* of x_2, x_1x_2 and x_2x_3 to z.

Localizer also includes an element constraint which complicates the calculation of invariants as just described: $element(e, x_1, \ldots, x_n)$ evaluates to the eth element of the x array. However, often it is very natural to write invariants such as $pos_1 = 1 + element(prev_1, pos_1, \ldots, pos_n)$. This example is taken from a TSP-based model and maintains the position (pos) of each town according to the position of the previous town. Unfortunately, this invariant is deemed illegal by the topological rule because the same expression (pos_1) is mentioned on the left and right hand sides of the equality and so the topological ordering constraint rule is violated. In Localizer, this is treated by so-called dynamic invariants. This means that some of the planning phase is delayed until the index of an element constraint is known by the execution phase. At that point, the element expression $element(e, x_1, \ldots, x_n)$ depends only on the eth x expression. So long as this is not the same as the left-hand side of the invariant (which would provoke an error), a planning phase is invoked to produce a topological order for the graph after (in a topological sense) this point. Planning and execution phases are interleaved as necessary to evaluate all invariants.

The Comet system [107,108] grew out of work on Localizer, but has some important differences. Localizer is mostly declarative in nature with some imperative parts for generating initial solutions. By contrast, Comet is a full-function imperative programming language with features oriented toward modeling and search. However, as for other constraint modeling systems embedded in an imperative language (for example, IBM ILOG CP Optimizer in C++, Java, or .NET), the way in which the constraint model is created is declarative in style. Unlike Localizer, Comet has some built-in machinery to evaluate constraint violations in a constraint

```
function void minConflictSearch(ConstraintSystem S) {
  inc{int}[] var = S.getVariables();
  range Size = var.getRange();
  while (S.violations() > 0)
    selectMax(i in Size)(S.getViolations(var[i]))
      selectMin(v in var[i].getDomain())
        (S.getAssignDelta(var[i], v))
          var[i] := v;
}
```

Fig. 3 A comet implementation of min-conflicts (from [109])

system, which allows the specification of the constraint model in more traditional form (through constraints rather than invariants).

Comet also has language features to iterate over, evaluate, and select different decisions, allowing a compact definition of search methods. Additionally, the impact of a potential move on the objective function can be evaluated using methods such as `getAssignDelta(var, value)` (change the value of a single variable) and `getSwapDelta(var1, var2)` (swap the values of two variables). Finally, mechanisms such as *closures* and *checkpoints* [107], allow complex control methods such as events and neighborhood evaluation to be specified more easily.

Figure 3 shows a small code (taken from [109]) which implements a min-conflicts heuristic using Comet. `S.getViolations` is used to select the variable which is involved in the most violated constraints and `S.getAssignDelta` is used to choose a value for it which reduces the violation measure by the largest amount (even if this amount is negative). This process continues until the total number of violations reaches zero, meaning a solution is found that satisfies all the constraints. Of course, as for min-conflicts, this state might never be reached.

2.4 Some Other Methods

Without aiming to be exhaustive, but rather restricting ourselves to the most CP-oriented methods (see [45] for additional references), we also very briefly mention some other techniques where LS has been applied to solve CP-based models.

In [111], Walser reports results on an LS technique on linear pseudo-boolean problems. In particular, excellent results are reported on the progressive party problem [11], which was considered a very difficult problem at the time.

In [21], Codognet and Diaz use a tabu search method on CSP models. Their method changes the value of a single variable at each move and uses a variable penalty which is computed from a combination of constraint penalties. Experiments are carried out on simple examples such as n-queens, magic square, number partitioning, and all-interval series. Results show that on some problem classes, the method outperforms other approaches.

Solnon [101] describes a method for solving CSP using an ant colony optimization approach. In Solnon's method, each variable assignment is represented by a node in a graph, and an arc is present between all nodes which correspond to different variables. After each ant completes its journey, its trajectory is improved by a classic LS algorithm. Additionally, a local search is used as a fast way of initializing the pheromone trails.

Galinier and Hao [32] present a method for constraint solving using LS. In particular, they specify a set of supported constraints (including global constraints) along with precise penalty functions (following [67]). They apply their method to SAT, graph-coloring, max-clique, frequency assignment, and the progressive party problem. The method is demonstrated to be very competitive.

In [86], Årgen et al. describe methods for performing LS over set variables with incremental updates of data structures. The authors apply their methods to the progressive party problem. In [85], the same authors also describe a method of synthesizing incremental update algorithms for LS from a description based on second order logic.

Finally, [36] shows how LS can be effective on quantified boolean formulas, which is a generalization of the standard NP models treated by CP.

3 Using CP to Evaluate Neighborhoods

One main area where CP and LS meet is when an essentially LS method uses CP to make and evaluate moves in the neighborhood of the current solution. Typically, such methods address optimization problems and maintain a complete assignment which satisfies all problem constraints, in contrast with many of the algorithms already presented which try to reduce constraint violations. These methods then move from feasible solution to feasible solution, in an attempt to improve the objective value.

These methods maintain the current solution not in the constrained variables themselves, but in a so-called passive representation which is a simple array s holding, for each variable, its value in the current solution. By contrast, the active representation is the constraint model itself with variables, domains, and constraints acting between them. When a new solution to be accepted is produced in the active representation, it is copied to s for safekeeping.

This section presents various methods for using CP to evaluate and make moves in an LS process. All the techniques may be implemented by traditional CP engines supporting a sufficiently flexible user-definable depth-first search.

3.1 Constraint Checking

In [5], De Backer et al. propose to integrate CP into an LS method in perhaps the simplest possible way. The CP system is only used to check the validity of solutions and determine the values of auxiliary constrained variables (including the cost variable), not to search for solutions (via backtracking, for example). The search is performed by an LS procedure. When the procedure needs to check the validity of a potential solution, it is handed to the CP system. The authors studied their integration in the context of vehicle routing problems.

When the CP system performs propagation and validity checks, it instantiates the set of decision variables with the proposed solution. The constraints then propagate and constrained variables (for example, any auxiliary variables whose values are taken from propagation from the decision variables) have their domains reduced. If any variable has no legal values, the proposed solution is illegal.

Improvement heuristics generally only modify a very small part of a solution. Therefore, testing the complete solution in the above manner can be inefficient. The authors try to avoid this inefficiency in two ways: by reducing the amount of work carried out by the CP system to perform each check, and bypassing the CP checks altogether. In the context of vehicle routing, it is often the case that routes are independent (have no constraints acting between them). For these problems, only the routes involved in the modification proposed by the move need to be instantiated in the active representation. Normally, this means just one or two routes will be instantiated.

The second method of increasing efficiency is to perform custom tests on the more important "core" constraints without using the general propagation mechanism [92]. Only if these fast checks succeed is the proposed solution handed to the CP system to be fully evaluated.

3.2 Exploring the Neighborhood Using Tree Search

In [73,74], Pesant and Gendreau propose a framework for LS in which a new "neighborhood" constraint model is coupled with a standard model of the problem, and represents the neighborhood of the current solution. The idea is that the set of solutions to this new combined model is the set of legal neighbors of the current solution. This setup is depicted in Fig. 4.

The left part of the figure shows the variables and constraints of the model of the problem (the *principal* model). On the right, there is another set of variables and constraints, which together represent a model of the neighborhood of the current solution. These variables are coupled to the principal model via *interface constraints* which ensure that a choice of neighbor in the "neighborhood model" enforces the appropriate values of decision variables in the principal model with respect to the current solution.

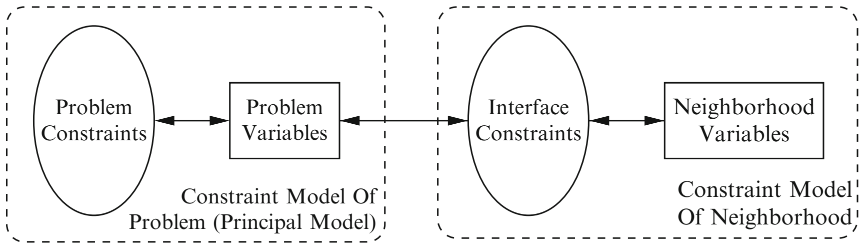

Fig. 4 Interaction of the neighborhood model with interface constraints

As a simple example, suppose that the principal model has n 0–1 variables $x[1] \dots x[n]$ and that we wish to perform an LS over a neighborhood which flips each variable (changes its value from 0 to 1 or vice versa). We suppose a passive representation of the current solution (as in the previous section) which is a simple integer array s of n elements. In this case, the neighborhood model would consist of a single variable f with n domain values (say $1 \dots n$). $f = a$ would mean that the ath 0–1 variable in the principal model would have a value different from that in the current solution, while the other constrained variables would take their values as in the current solution. The interface constraints are as follows:

$$(f = i) \Leftrightarrow (x[i] \neq s[i]) \ \ \forall\, i \in \{1 \dots n\}$$

To explore the neighborhood of the current solution, a backtracking search procedure then finds all solutions to the combined model, normally through exploring all combinations of values of the neighborhood variables: here, this means instantiating just the variable f (so a search tree of depth 1). This exploration is very natural in a CP context–one may also select the best (lowest cost) neighbor to move through a standard branch-and-bound search over the model.

It should be noted that the neighborhood model is configured according to the current solution vector s. That is, constants in the neighborhood model are taken from the values of the current solution. This means that, in general, the neighborhood model is recreated anew at each move, the previous one being discarded.

3.2.1 An Example

We give a more realistic example of a complex move which may be implemented using the constrained neighborhood framework. Consider an allocation problem, like a generalized assignment problem, where n objects are allocated to m containers. This problem might have capacity constraints on the containers, a complex cost function, and other side constraints, but that is not our concern here–we are only interested in how a solution is represented and the decision variables. Our passive representation of a solution will store the container s_i into which object i is placed. Our principal constraint model will contain n decision variables $x[1] \dots x[n]$ each with domain

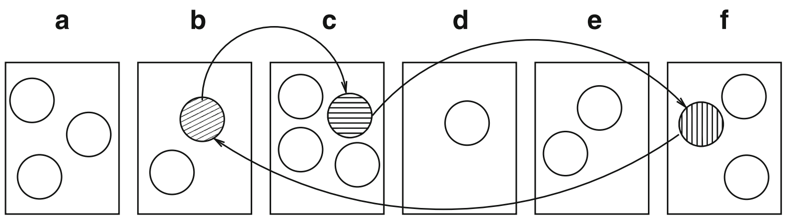

Fig. 5 A 3-rotation move

$\{1 \ldots m\}$. The neighborhood we consider is a rearrangement of the positions of any three objects without changing the numbers of objects in each container. We call this a 3-rotation. The move is depicted in Fig. 5. In the figure, after the move, the diagonally striped object will have moved to container C, the horizontally striped object to container F, and the vertically striped object to container B.

We will create a neighborhood model with three variables p, q, r, representing the objects involved in the move. Each of these variables has domain $\{1 \ldots n\}$ identifying the objects to be involved in the move. We will assume that in the move, object p will move to q's container, q to r's container, and r to p's container.

The interface constraints for this neighborhood can be written as:

$$p \neq q, p < r, q < r \tag{1}$$

$$x[p] = s[q], x[q] = s[r], x[r] = s[p] \tag{2}$$

$$p \neq i \wedge q \neq i \wedge r \neq i \Rightarrow x[i] = s[i] \;\; \forall\, i \in \{1 \ldots n\} \tag{3}$$

Line (1) constrains which objects can be involved in the move. Some symmetries need to be broken if moves are not to be duplicated. For example, $p = 1$, $q = 2, r = 3$ identifies the same move as $p = 2, q = 3, r = 1$. These can be broken by the constraints $p < r, q < r$. Note also that three *different* objects should be identified and so $p \neq q$. Line (2) places the objects involved in the move in their new positions. For this, the element expression is used on the solution vector s. Line (3) states that objects not involved in the move stay where they are currently.

Observe that whenever two objects appear in the same container, the move actually taken (so long as the third object is elsewhere) is a swap of two objects. If desired, we could limit ourselves to true 3-rotations by asserting that all objects must be in different containers. This could be done by replacing line (1) above by:

$$s[p] \neq s[q], s[p] < s[r], s[q] < s[r]$$

3.2.2 Flexibility and Efficiency

The main advantage of this framework is that propagation takes place between the principal and neighborhood model, and specifically, domain filtering may be carried out on the neighborhood variables, reducing the number of neighbors explored. This filtering is carried out by the interface constraints, based on the current domains of the decision variables of the principal model. The final result is that neighbors which are illegal or do not meet acceptance criteria (like being better than the current solution in terms of cost) may be directly filtered out of the neighborhood variables. This filtering tends to be most effective when the search tree for the neighborhood variables is deep (neighborhoods are large) and the neighborhood has a large proportion of illegal neighbors. Experiments with the 3-opt operator on TSPs with time windows in [73] indicate that few neighbors are actually acceptable (typically a few percent), increasing the efficiency of the CP technique. GENIUS-CP [75] is very general technique for inserting customers into a route, and implemented in the CP framework. The authors report that one does pay a price for using CP in terms of efficiency (up to an order of magnitude over a custom approach), but their technique extends naturally to TSPs with multiple time windows and pickup-and-delivery problems, which the custom heuristic [35] was incapable of doing. In [91], the flexibility of the method is demonstrated via the application of several different constraint-based neighborhood operators in a style based on variable neighborhood search (VNS) [65].

Although the above framework is extremely flexible, natural, and exploits the power of CP's propagation in order to reduce neighborhood sizes, this benefit is not obtained when either the neighborhood is largely feasible, or when opportunities for propagation are limited (for instance when the depth of the search tree on the neighborhood variables is low). For example, in the "flip" example already given, exploring the neighborhood through a single depth search tree which instantiates the f variable could result in a total of n^2 variable fixings.

To attempt to address this, Shaw et al. [100] describe a method which reduces the number of variable fixings to $O(n \log n)$ for the "flip" example. The method works on an explicit representation of the neighborhood as a series of neighbors, each neighbor represented by an object called a "delta" which stores the variables to change together with their new values.[3] One attraction of this method is that it allows the creation of very general, unstructured, or randomly ordered neighborhoods, specifiable via general codes, and not constrained to what can be described by a constraint model. For this reason, this method is used in an open framework in the Solver module of IBM ILOG CP (which provides operators to randomize, concatenate, interleave and sample general neighborhood structures).

Figure 6 shows an actual C++ code sample for solving a vehicle routing problem using the IBM ILOG CP product. This code uses a greedy approach (via the use of `IloImprove`[4]) which accepts the first neighbor which decreases the overall cost. (By adding one additional line of code, it is also possible to accept the neighbor which decreases cost by the greatest amount.) The neighborhood is a combination of all possible two-opt [59] moves, all moves which relocate every node before each other node, and all moves which interchange the positions of two nodes.

[3] Not all neighbors have to be available before exploration can start, but can be generated on the fly to keep memory consumption low.

[4] Meta-heuristics such as simulated annealing or tabu search can be used by changing `IloImprove` to `IloSimulatedAnnealing` or `IloTabuSearch`.

```
// env = environment, a management object
// curSol = current solution
// solver = the CP solver instance
IloNHood nh = IloTwoOpt(env) + IloRelocate(env) + IloExchange(env);
IloNHood rnh = IloRandomize(env, nh);
IloGoal move = IloSingleMove(env, curSol, rnh, IloImprove(env));
while (solver.solve(move))
  cout << "Objective value = " << solver.getObjValue() << endl;
```

Fig. 6 IBM ILOG CP example of local search for vehicle routing

The "+"operator is used to concatenate these three different basic neighborhoods into one. This simple but powerful method can be used to implement a VNS [65]. Here, instead of using a VNS method (which could be achieved by using `nh` instead of `rnh` in the call to `IloSingleMove`), the order of moves in the neighborhood is randomized each time a move is accepted (by using `IloRandomize`), so that a *random* improving move will be taken each time. A loop calling `solve` on the CP solver actually performs the greedy moves. When no improving move exists in the neighborhood, the call to `solve` returns a false value, and the loop exits with the locally optimal solution in `curSol`.

The method described in [100], and used in IBM ILOG CP, gains efficiency by using a divide and conquer approach: a binary search tree is created where on the left branch the first half of the neighborhood is explored, and on the right branch, the second half is explored. This is continued recursively until only one neighbor remains at the leaf node. Each node in the search tree can thus be associated with a part of the complete neighborhood. If this part of the neighborhood does not change a variable x (that is, x does not appear in any "delta" for this part of the neighborhood), then the value of x in the current solution s can be assigned to x. This last rule, which allows different neighbors to share variable fixings, reduces the number of fixings by a factor of $O(n/\log n)$ when neighbors change a constant number of variables.

Pesant and Gendreau also propose an adaption of their method which allows similar logarithmic efficiency gains to be achieved. The idea is to map the divide and conquer approach of Shaw et al. to a variable instantiation strategy. Instead of instantiating the neighborhood variables by fixing them, a deeper search tree can be created by dividing the domain of a neighborhood variable in two at each branch. This will eventually instantiate the neighborhood variables but will crucially allow propagation to the principal model's decision variables higher in the (now deeper) search tree.

3.3 *Large Neighborhood Search*

Large Neighborhood Search (LNS) is a technique first coined in [99], but whose origins date back to the shuffling technique described in [3]. Similar work uses different nomenclature: Mimausa [60], forget-and-extend [17] and ruin-and-recreate [94]. In [40], Sect. 10.7 also briefly mentions such an approach in general terms, there called *referent-domain* optimization, and [104] proposes a technique with similar motivation.

LNS has been applied to a variety of problems, with most of these having a vehicle routing [10,76,99], scheduling [3,41], or a network/graph aspect [15,20]. It is also used in commercial products, such as IBM ILOG CP Optimizer [55].

The simplest way to think about LNS is that of iteratively relaxing a part (normally called the *fragment*) of the current solution and then re-optimizing that part. Figure 7 shows how this method works for the traveling salesman problem (TSP). Figure 7a shows a solution to the TSP, in Fig. 7b,c a set of arcs is chosen and removed from a region of the space, and then in Fig. 7d this partial solution is completed to produce a new, improved solution of lesser total distance.

An outline of how LNS is normally implemented is shown in Fig. 8 (adapted from [72]). At a high level, the technique is extremely simple. It is normally a hill climber (however see [76]) which is executed until some time limit is exhausted, although other criteria can be used, such as a number of iterations, or a number of iterations without improving the current solution.

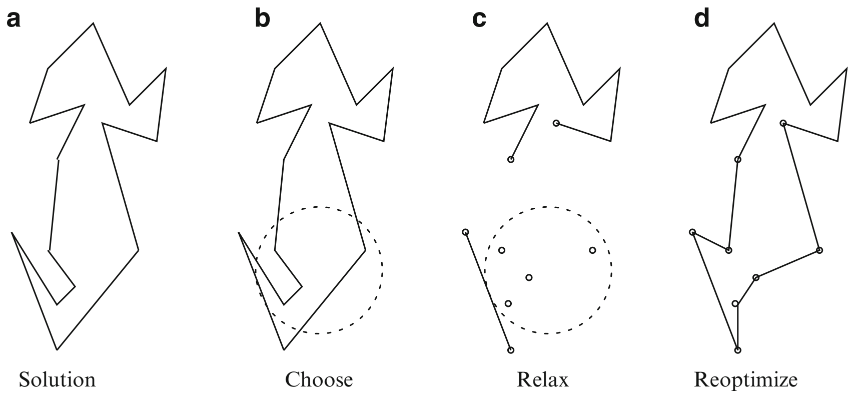

Fig. 7 Operation of LNS on the traveling salesman problem

```
Create initial feasible solution
while optimal not found and time remaining do
    Choose part of problem (the fragment)
    Freeze the part of the solution not in the fragment
        (that is, fix it to its value in the current solution)
    Search the remaining sub-problem for an improving solution
    if improvement found then
        Adopt the improved solution as the current one
    end if
end while
```

Fig. 8 Typical implementation of large neighborhood search

The general description of LNS makes reference to two components: a method to choose the fragment of the problem to relax, and a method to re-optimize the chosen fragment: we discuss both, beginning with the latter.

3.3.1 The Re-Optimization Method

LNS is specified in very general terms, and this leaves the implementation of the re-optimization method open. Techniques which have already been used with LNS are greedy methods [76], LS [103], decomposition-oriented techniques [68], mixed-integer programming methods [79] (for linear models), or CP [99]. In fact, virtually any technique which can be used to solve a complete problem can be used to re-optimize a fragment in LNS. However, for CP techniques, the re-optimization is generally based on a tree-search, which could be depth-first search, limited discrepancy search (LDS) [44], restarts [52], or indeed any other techniques one might imagine. Using CP as the re-optimization method allows the addition of a constraint on the objective which excludes non-improving solutions. This can accelerate the search through increased propagation.

Using CP for the re-optimization process also increases the robustness of the search as side-constraints can be efficiently handled in the re-optimization, meaning that LNS is much better able to handle richer models with more complex constraints than standard neighborhood search. This is demonstrated in [53] where side constraints are progressively added to a problem, and in [9], which successfully extends an LNS solution for vehicle routing [8] to one for pickup-and-delivery, which is considerably more constrained. Likewise, Røpke and Pisinger [76] and Laborie and Godard [55] introduce LNS methods which perform well over a wide range of vehicle routing and scheduling problem classes.

When re-optimizing the fragment, we have found that a tree search which works better than others for solving the *whole* problem will continue to work better than others when embedded in LNS as a re-optimization procedure.[5] We know of no strong contradictions of this rule of thumb–if any exist, their analysis might yield interesting directions of the design of LNS re-optimization methods. That said, an almost ubiquitous modification is made to any global solution method when it is embedded in LNS, and that it to limit its run time in some way ([13] is an exception). Common methods are to limit the number of backtracks in a depth-first search [16, 71], or to limit the number of discrepancies [6] in a depth-first search [99]. Perron [70] compares both of these methods, amortized search [58], as well as hybrids.

The general idea is to find a balance between avoiding exponential time exploration of unpromising fragments, and attempting to improve the solution as much as possible at each re-optimization. Another choice in the re-optimization is whether

[5] This is important, as here we can draw on a wealth of knowledge on general-purpose and problem-dependent branching heuristics.

to stop as soon as an improving solution is found (as done in [71]), or to continue up until the limit to see if better improvements can be found in the same re-optimization (as done in [13, 99]).

3.3.2 Selecting the Fragment

One natural way of selecting the fragment of the solution to be relaxed is to do this in a completely random fashion. This technique can sometimes be effective. For example, [69] uses a completely random technique (but then moved to a more structured technique in [70]), [23] shows good results with random selection (although biased selection is shown to perform better), and [71] uses a random selection as one of two methods for fragment selection. Although a random choice has its merits (it may avoid pathological behavior, and is thus useful in portfolio-based techniques– see Sect. 3.3.3), using *only* random choice is seldom the best way to produce good solutions quickly. This was recognized from early on, for example, the shuffle of [3] relaxes the order of all operations on a set of machines–each machine has either a fully fixed or fully free order (while still respecting problem constraints). Later, other more general shuffles were performed in [16], but here again the fragments chosen have a certain cohesion, for example, operations falling in a certain time window, or operations on the critical path.

We should ask ourselves the question, *what makes a good fragment*? To be useful, a fragment must contain at least one good solution. (In the typical implementation of LNS, this solution should be better than the current solution, but some implementations allow cost-neutral moves [71] or degradation of the current solution [76].) For some problems, we can determine some basic conditions that need to be satisfied if cost is to have a hope of being reduced. For example, in a job-shop scheduling problem with a makespan objective, we know that the critical path must be changed to reduce the makespan. This dictates that some operations on the critical path should be present in the fragment, if the fragment is to be useful [13].

Other less strict rules can be used to select the fragment which we might consider would generate fragments with improving solutions. For example, Fig. 7 shows a fragment selection criterion based on distance from a randomly chosen node n in the TSP tour: for example, one might relax the k closest nodes to the central node n. The intuition here is that when we relax the fragment, the relaxed nodes will have the possibility for position interchange in the current tour. Now, assuming the current tour is not too bad (if it is bad, then almost any reasonable fragment selection will allow improvements to be found), then nodes which are far apart are also far apart in the tour. This implies that interchange of these nodes would introduce long arcs, and hence a large distance penalty. By contrast, when the fragment is localized, interchange of nodes can be carried out without introducing long arcs.

Given the fact that a fragment can be selected that contains a better solution– or has a reasonable chance of containing one–the fact that the solution is available is not enough. (We could place the whole problem in the fragment, which would guarantee it contained the global optimum.) In reality, the re-optimization method

```
(Assume solution elements are $x_1 \dots x_n$)
Initial fragment $F = \{i\}$ where $i$ is randomly drawn from $\{1 \dots n\}$ ($F$'s universe)
while $|F| < f$ do
    Let $e$ = random element of $F$
    Let $z$ be an non-increasing ordering of all $e' \in F^C$ according to $R(s, e, e')$
    ($F^C$ is the complement of $F$, the indexes of the elements not in the fragment)
    $k = 1 + \lfloor |F^C| y^d \rfloor$ where $y$ is uniformly and randomly chosen in $[0 \dots 1)$
    Add the $k$th element of $z$ to $F$
end while
return $F$
```

Fig. 9 Shaw's method for choosing a fragment

must have a reasonable chance of finding the solution using the limited resources at its disposal. There is thus an interplay between the fragment selection and the re-optimization method, but the general idea is to keep the fragment as small as possible, while maintaining a reasonable chance of finding an improving solution. In this way, the re-optimization process has the best chance of finding the good solution or solutions. Similarly, smaller fragments allow the re-optimization process to be limited more, which increases the speed of examination of different fragments. A common mechanism, first used in [99] but since replicated elsewhere, is to begin with a small fragment, and gradually increase its size when improving solutions become difficult to find.

Although the use of some problem structure is useful for choosing the fragment, the first generic framework for doing this was introduced in [99]. We outline the approach in Fig. 9. It assumes the existence of a real valued function $R(s, x, y)$ which takes the current solution s, and two solution elements. The solution elements could be variables, but could also correspond to more structured objects such as a nodes in a TSP or operations in a scheduling problem. R delivers a measure of how related x and y are in solution s. The interpretation of this is that if $R(s, x, y)$ is high, then x and y should tend to be included in the fragment together, or not at all, and if $R(s, x, y)$ is low, then putting both x and y in the fragment is less desirable. In [99], vehicle routing problems were treated, where an objective which was to reduce the number of vehicles used, and then total distance as a secondary objective. For two customers x and y, $R(s, x, y) = 1/(C(x, y) + V_s(x, y))$ where $C(x, y)$ is a normalized distance (in the range $0 \dots 1$) between the two customers x and y and $V_s(x, y)$ evaluates to 0 if the two customers are served by the same vehicle in the current solution s and to 1 otherwise.[6] The idea of favoring customers served by the same vehicle for inclusion in the fragment comes from the observation that the only way of reducing the number of vehicles (primary objective) is to free all customers on at least one vehicle. Unless some specific provision is provided for that, the distance measure alone is unlikely to achieve it.

[6] An error appeared in the original paper which indicated the opposite result for $V_s(x, y)$.

The selection method also has two parameters $f \geq 1$ (fragment size) and $d \geq 1$ (determinism). The former indicates the desired size of the fragment, and the latter a measure of how deterministically the selection will follow R: $d = 1$ means that R will be ignored and random choice will be used, while $d = \infty$ means that R will be followed for each choice.[7] Shaw [99] reports that the tuning of the determinism parameter was reasonably simple, and for medium sized VRPs (around 100 customers) values around 10–20 were reasonable, with performance suffering badly only for values of d outside $3 \ldots 30$. Pisinger and Røpke [78] also discuss fragment selection methods based on the distance between customers in a vehicle routing context. They use Kruskal's algorithm [54] to generate clusters of customers which are close together.

3.3.3 Learning in Large Neighborhood Search

A few ideas have been put forward to attempt to ease the design and tuning of LNS methods (see discussion in [14]). The first is the aforementioned one of starting with a small fragment size and increasing it when finding improving solutions is becoming too hard. This is normally controlled by a number of LNS iterations without an improvement in the current solution.

In an attempt to alleviate the work involved in designing fragment selection, Perron and Shaw [72] propose a method called Propagation Guided LNS which tries to determine what solution elements should be placed in the fragment together. The method works by using the propagation mechanism itself. It can be viewed as a variation on Fig. 9 where each time a solution element is added to the fragment, it is assigned as in the current solution s, and propagation on the sub-problem outside the fragment is observed. The strength of the propagation of x to y (where x is newly added to the fragment and y is outside the fragment) defines $R(s, x, y)$. The method continues like this until the desired fragment size is attained, at which point it records the fragment for use in the subsequent re-optimization process, but discards the partial assignment corresponding to it in the constraint solver. Perron and Shaw show that the method can outperform hard-designed fragment selection on the car sequencing problem.

A common learning mechanism applied in the context of LNS is that of algorithm portfolios [43]. LNS is particularly amenable to such a technique as it is an iterative combination of two separate techniques: fragment choice and re-optimization. Thus, a portfolio of algorithms can be proposed for each technique. In [70], Perron uses a portfolio of algorithms for the re-optimization technique for a network design problem. He ascribes a weight to each algorithm and uses reinforcement learning to reward (increase the weight of) algorithms which produce improving solutions. The weights drop after a certain number of trials without improvement. An algorithm

[7] Setting $d = \infty$ does not eradicate randomness from the algorithm, only for one part of it.

is selected proportionally according to its weight in relation to others. However, Perron's main aim in [70] is to investigate restart strategies, and does not perform any detailed analysis of his portfolio scheme.

Røpke and Pisinger [77] present a method which uses portfolios for both the fragment selection method and the re-optimization method for pickup-and-delivery problems. They use three fragment selection methods, five re-optimization methods and optionally apply noise to the objective function to help diversify the (largely greedy) re-optimization method. A feedback learning approach controls a biased choice of fragment selection, re-optimization method, and the application of noise to the objective function. Their fragment selection methods are based on that of [99], a completely random method, and a method which tries to place high-cost solution elements into the fragment (reminiscent of squeaky wheel optimization [47]). Re-optimization methods are founded on a regret-based technique. The authors method demonstrates greatly increased robustness over a standard LNS approach (allowing them to develop a very general heuristic applicable to many different types of routing problem [76]), and produces state-of-the-art results on pickup-and-delivery problems with time windows.

Laborie and Godard [55] also present a portfolio method and apply it to a large number of different single-mode scheduling problems. Their framework also proposes using a feedback learning approach on fragment selection and re-optimization methods, but their study is limited to the selection of fragment. Three fragment selection methods are used: random selection, selection of operations in a time window, and selection of operations occurring in connected components or strongly connected components of the precedence graph of the problem. Results show that the resulting algorithm is extremely robust over a large number of scheduling problem classes and, despite its generality, can improve on the best ad-hoc methods created for several of these classes.

In [18], Caseau et al. present a general method for learning combinations of heuristics for vehicle routing problems. They use LNS with fragment selection as in Fig. 9 as one of the building blocks of their system. Their LNS method is parametrized with a learned fragment size f, a learned determinism parameter d, and a learned re-optimization method, which can be constructed from other basic building blocks. Results show that the learning process can better hand-crafted algorithms.

4 Local Search for Pruning and Propagation

LS has been used as a means of demonstrating that the current partial solution to a problem has no legal extension (resulting in pruning) or that certain value assignments cannot be made in any legal extension (resulting in domain filtering, or propagation as it is also known). Although this area has not been widely explored, this section describes some pieces of work which have successfully used LS to reduce search tree size through pruning and/or propagation.

In [96], Harvey and Sellmann describe how to increase pruning and propagation on the Social Golfer Problem. The social golfer problem can be stated as follows: *32 golfers want to play in 8 groups of 4 each week for 9 weeks, such that no pair of players play together more than once. Is this possible?* The problem can be generalized to one of determining a w week schedule of g groups of golfers, with each group of size s. Harvey and Sellmann's method depends on finding witnesses to the insolubility of the current sub-problem. These witnesses are clique structures contained in a residual graph representation of the sub-problem.

Consider that a golfing schedule is being built week by week, in chronological order, and that a graph is maintained in which each golfer is represented by a node and a pair of golfers that have already played together in previous weeks is represented by an arc connecting their respective nodes. If a clique of size k is present in this graph, then it means that the golfers involved in the clique must play in different groups from now on. If, when scheduling the current week by building groups for that week, we get to a situation where none of the golfers in the clique have been scheduled, but less than k groups remain available this week, then the current partial assignment cannot be completed as each of the k golfers in the clique must be put into a different group. Harvey and Sellmann use an extension of this pruning rule to perform propagation and describe an orthogonal rule which reasons on golfers rather than on groups in a week. The authors use a randomized LS approach with intensification and diversification steps to find the cliques. Results show that significant gains can be made both in terms of number of backtracks and run time.

Focacci and Shaw [30] present a method for pruning the search tree based on the application of a dominance rule, the detection of the rule's applicability at any node being carried out by an LS process. The method could be considered an implementation of Symmetry Breaking by Dominance Detection [26, 29]. Focacci and Shaw argue that aggressive pruning of a search tree by dominance rules can be detrimental to the success of a search algorithm. For example, consider a Euclidean TSP where a 2-opt [59] dominance rule is applied. That is, any branch of the search tree which constructs a partial tour where two arcs cross (and is therefore not 2-optimal) is pruned, as we know that any tour which is not 2-optimal cannot be globally optimal. This aggressive pruning rule may result in no solution being found for some considerable time, which could be a problem for online optimization. Furthermore, since solutions may be found considerably later with the dominance rule active, even time to produce a proof may suffer since good upper bounds (which can also significantly help pruning) are not provided quickly. The solution proposed in [30] is quite general (a similar technique is used in [31]) and can be applied to decision as well as optimization problems, but the following description is based on the TSP.

The authors propose to apply a dominance rule which prunes the current branch only if there is no hope of extending it to a solution which can better the best solution found so far. This is done in the context of the TSP and TSP with time windows. Here, imagine that we are to construct a simple path between nodes A and Z, passing through nodes $\{B, C, \ldots, Y\}$. Imagine two partial TSP tours $t_1 = A \to B \to C$ and $t_2 = A \to D \to B \to E \to C \to F \to G$. Assume that all extensions of t_1 have already been fully explored and that t_2 is the current partial tour. We form t

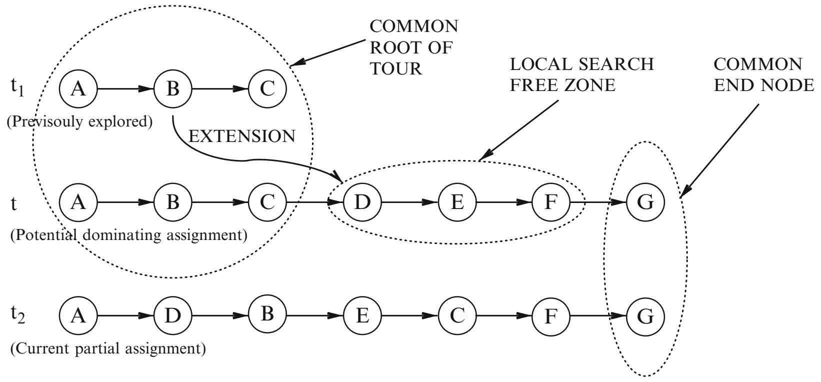

Fig. 10 Operation of Focacci and Shaw's dominance detection

from t_1 by appending to it all nodes in t_2 not already in t_1, ensuring that t ends at the same node as t_2, node G in our case.[8] Assume that after performing this operation $t = A \to B \to C \to D \to E \to F \to G$. Observe that with this construction, t always visits at least all the nodes in t_2 and terminates at the same node. Figure 10 shows the relationships between the three partial tours. Now, assume that the length of a (partial) tour is given by the function L and that in our case $L(t) \leq L(t_2)$. Given that t visits at least all of the nodes of t_2 and any extension of t will start at the same node as t_2, we can safely say that the best extension of t will be better than the best extension of t_2. However, we have already explored the best extension of t, because we have explored the best extension of t_1 (a smaller partial tour). Thus, we can prune the current search tree branch t_2. If $L(t) > L(t_2)$, then we still have options. In particular, we formed t from t_1 by an arbitrary extension order. Any extension which maintains G at the end of the partial tour is legal to form t. If any one of these extensions verifies $L(t) \leq L(t_2)$, then search can be pruned.

Focacci and Shaw propose to search for an improving extension t via an LS process which, in the case of our example, would perturb the order of the nodes D, E, F, while keeping G at the end as needed, as well as the root of the partial tour (ABC) coming from t_1. This is done using a relocate operator which can move any node in the free part of the partial tour to another position in the free part (see Fig. 10). The extension to the TSP with time windows simply adds another condition to check. Assume that the total time of a (partial) tour is given by the function T. Then, search can be pruned if $L(t) \leq L(t_2) \wedge T(t) \leq T(t_2)$.

Results reported indicate that the proposed approach can often work much better than a pure CP approach without dominance rules or indeed than standard application of dominance rules which prune every branch which corresponds to a partial tour that can be improved using the relocation operator.

[8] If t_1 already contained G, then the dominance detection would not be possible using t_1.

Galinier et al. [33] describe a method to facilitate propagation of the *some-different* constraint [90]. The some-different constraint is a generalization of the all-different constraint [88] where only a subset of all pairs, freely specifiable to the constraint, must be different. This global constraint is capable of encoding any graph coloring problem and hence achieving arc consistency for it is NP-hard.

The authors approach the problem by using a tabu search algorithm to find *supports* (solutions to the constraint). The aim is to quickly find a support for each domain value of each variable. For those variable-value assignments for which a support cannot be easily found, the suspicion is that they have no support, and this can be found out by the application of a complete technique, such as DSATUR [12]. The LS algorithm is reminiscent of min-conflicts but is based on tabu search [40] and works as follows. Initially, each variable is assigned an arbitrary value from its domain. Then, the search attempts to minimize the number of conflicts ($\neq$ constraints violated) using a tabu criterion which forbids a variable taking on its previous value for a certain number of iterations. When a solution to the constraint (a support) is found, then it is added to a "support store".

After the first support is found, the algorithm changes its behavior, as it is no longer based solely on reducing the number of conflicts, but a weighted combination of conflicts and on covering (in the support sense) variable-value assignments not yet represented in the support store. In this way, the tabu search algorithm tries to provide a support for as many variable-value assignments as quickly as possible. The tabu search algorithm is stopped after a certain number of iterations after which it is considered that continued search is not as productive as the use of other methods. After the tabu search has found as many supports as possible, a complete method (DSATUR) is launched for each variable-value assignment for which a support has not yet been found. This is done for $x = a$ by assigning a to x and then using DSATUR to try to find a solution to the remaining sub-problem. If this sub-problem proves soluble, then a support for $x = a$ has been found, otherwise a is removed from the domain of x.

Combined with a number of reduction procedures, Galinier et al. report that their algorithm reduces average run time on real workforce management data by a factor of 34 over a standard CP approach.

5 Other Integrations

Various other integrations of LS in CP have been proposed where the interaction of the CP and LS techniques is more intimate. Often, the methods are based on the combination of a constructive technique (including propagation) and a repair technique, but could simply be the use of a non-systematic search idea inside a complete search method. Much work could potentially fall into this fairly weak classification, and so while this section is necessarily not exhaustive, it does attempt to present a reasonable overview of the current situation.

In [114], Yokoo describes a *weak commitment* strategy for solving constraint satisfaction problems. The algorithm is complete and the main idea is based on the complete min-conflicts method [64]. The complete min-conflicts method is essentially a depth-first search using a min-conflicts heuristic for variable and value choice. First, a tentative assignment to each variable is made which greedily attempts to minimize total conflicts as described in Sect. 2.1. Then, a depth-first search chooses first a variable in conflict, assigning it a value which does not conflict with previous assignments and conflicts minimally with future tentative assignments. If all values conflict with previous assignments, the algorithm backtracks.

Yokoo observes that in depth-first search mistakes are very costly to undo. Yokoo's weak commitment strategy also uses tentative assignments, but instead of backtracking one level when a dead end is reached, it entirely abandons the current partial assignment. Search then continues by building a new partial assignment, using, where possible, the previous assignment as new tentative assignment values and adding the previous abandoned assignment to a no-good store to avoid re-exploration of the same space. One can think of the weak commitment method as a type of improved *iterative sampling* [56], where the variable/value choice heuristic is based on min-conflicts, and a no-good is added at each dead end.

Yokoo observed good results on graph coloring and 3-SAT problems, better than either the complete min-conflicts method or the breakout method.

Richards and Richards [89] describe Learn-SAT, whose principles draw heavily on the weak commitment method. Their algorithm uses the same process of abandoning the search process when a dead end is encountered, and of no-good addition. However, Learn-SAT also including a "learning-by-merging" component which performs resolution on the clauses which lead to a variable's domain becoming empty, generating a new clause for each dead end encountered. Experiments demonstrate that the method can outperform *relsat* [48], the best complete SAT solver of the time.

Ginsberg and McAllester [39] describe how gradient methods from GSAT can be used in a complete technique. The method used, known as partial order dynamic backtracking (PDM), works by generating no-goods in *directed form* ($A = a \wedge B = b \wedge C \neq c \Rightarrow D \neq d$ instead of $A \neq a \vee B \neq b \vee C = c \vee D \neq d$) when a violation is detected. The value of right hand side (conclusion) variable is then modified by the algorithm in order to remove the violation. PDM allows some freedom in the choice of the conclusion variable of any new no-good generated. This freedom allows PDM to start from a tentative total assignment, and then use a GSAT-inspired conflict-based rule to repair (that is, to add as conclusions in a no-good set) the variables which are most in conflict. In this way, a complete search is created, based on information normally only available to an LS technique.

It should be noted, however, that the choice of next variable to repair is not as free in PDM as in GSAT. PDM introduces so-called *safety conditions* to guarantee integrity of the algorithm, which specify partial orders on the variable choice. Ginsberg and McAllester demonstrate that their methods compare very well with WalkSAT and TABLEAU, dominating TABLEAU on unsatisfiable SAT instances.

Jussien and Lhomme [49] introduce the decision-repair algorithm which combines a constructive search and propagation with an LS over the decisions made in the constructive search path. This incomplete method works by constructing a solution using heuristics and constraint propagation in the "classic" way until the last decision D made creates an inconsistency. Instead of undoing D in the normal depth-first search manner, decision-repair tries to find a *conflict*, which is a strict subset of the decisions already taken (not including D) which are together inconsistent with D. At least one of these decisions needs to be negated if D is to be kept. The authors introduce a heuristic to decide which decision(s) to negate based on a tabu search criterion. The most recent k conflicts found during the search are maintained, and the algorithm first chooses to negate a decision which appears most often in this conflict set. If more decisions need to be negated to ensure D can be accepted, then these follow the same rule. Experiments on open shop scheduling problems show the technique to be competitive with leading approaches.

Prestwich [80] introduces CLS (constrained local search), later to be renamed IDB (incomplete dynamic backtracking) [83]. Like [49], the method is incomplete, and can be considered as performing an LS on the search decisions already taken in a tree search. The difference between decision-repair and IDB is based on how the repair of past search decisions is carried out. In [49], conflicts are used both as a predicate and as a heuristic to decide on the decisions(s) whereas IDB either uses random selection, or an ad-hoc heuristic with random tie-breaking. Once the repair variables are chosen, in [49], each repaired decision is negated, whereas IDB simply discards the chosen decisions. Prestwich has applied IDB quite widely with excellent results: see for example [80–82]. In particular, in [81], Prestwich argues that IDB delivers the scalability of LS compared to classic backtracking *because* of the flexible choice of the variable to change.

In [95], Sellmann and Ansótegui describe Disco-Novo-GoGo, a restarting technique which uses backtracking search "probes" to perform an LS over a complete assignment. The essential idea is to encode a value choice heuristic as a complete (normally illegal) assignment H. A backtrack-limited depth-first search then instantiates the problem variables in a random order, fixing each variable, where possible, to its preferred value from H: this is done on the left branch of the search tree, whereas other values are considered equivalent and are explored in random order on other branches. If the backtracking search finds a solution or proves there is none, the entire search process can be stopped. If the backtracking limit has been attained (which is usual), then the heuristic assignment is updated as follows. For any variable which is fixed to a different from its value in H, its current value is copied to H. For any unassigned variable that has had its value from H filtered from its domain, a random value from its current domain is transferred to H. The remainder of H remains unchanged. This basic mechanism is embedded in a double loop which varies the fail limit and a number of trials. The authors report significant speedups over standard restart techniques.

Zhang and Zhang [116] report on a method which combines an LS procedure with backtracking and propagation. The idea is to successively generate, in a randomized way, partial solutions to the problem which do not violate the constraints between variables in the partial solution. Then, for each one of these partial solutions, a combined LS and backtracking phase is entered where first a tree search technique with propagation tries to extend the partial solution to a complete one. This tree search is normally limited in the number of branches it can carry out. If the tree search does not succeed, a local move is carried out on the partial solution to try to improve its quality. The authors use the criterion of the number of constraints satisfied by the partial solution. This extension attempt using propagation and backtrack followed by a local movement on the partial solution is repeated until a solution is found or an iteration limit is exceeded. In the latter case, the method begins again with a completely new partial solution. The authors succeed in closing some open quasi-group problems using their method.

Schaerf [93] describes an incomplete method. The technique extends a partial solution until a dead end is reached. (A static variable ordering is used and Schaerf defines a dead end as being when the next variable to be assigned has no legal values consistent with previous assignments.) At this point, an LS method is invoked which attempts to change the values of the variables in the current partial assignment to allow further constructive search to take place. The method continues in this way, interleaving construction and repair, until a solution is found.

When repair takes place, the objective function takes into account (a) conflicts existing between variables in the partial solution, (b) the value of the bound on the objective function, and (c) a "lookahead" which is a count of the number of domain values remaining in the variables not yet given a value. The idea is to reduce (a) and (b), but increase (c). The LS procedure is largely greedy but is randomized and may make sideways (non-objective changing) moves. In addition, the different constraint types in the problem have independent and dynamically changing weights according to how often they are violated.

The algorithm is evaluated on timetabling and tournament scheduling problems. Schaerf compares LS (for example min-conflicts) to a combination of the LS method embedded (and suitably modified) in the constructive procedure. Results show that the combined method significantly outperforms the LS method on these problems.

At around the same time, David [25] also introduced an incomplete method very similar to that of Schaerf. A partial solution is extended until a dead-end is reached, at which point action is taken to try to repair the current partial assignment. David concentrates on examination timetabling problems, and the repair methods are dedicated to the problem. Four different repair techniques are tried before the method marks an examination as "not scheduled," and continuing with the remaining examinations. Thus, the method may produce partial schedules (which are presumably executed by temporarily procuring extra staff), but produces results in a deterministic time frame (under a few seconds).

Caseau and Laburthe [19] present an incremental local optimization approach to solving vehicle routing problems where a traditional constructive approach to building a solution is followed, but the partial solution is improved each time the solution is extended. The authors begin with an empty schedule and use a classical insertion approach to solution construction. At each insertion, the most "constrained" customer is inserted. Before insertion, a limited search examines the potential impact of insertion in different routes and the best insertion point is chosen. After insertion, a more thorough LS process improves the solution. Results show that the method scales much better than performing all insertions and then applying LS. The authors find very good solutions to problems with around 500 customers in around 10 s. Finally, to further improve their solutions, the authors also examine overlaying a limited amount of backtracking over their methods. For example, the decision to open a new route or not when inserting a customer could be the subject of a backtrack point.

Kamarainen and El Sakkout [51] describe a method called local probing, a type of probe-backtrack search [50]. In probe-backtrack search, at each node in a tree search, a *probe* is executed which tries to find a solution. This probe is some heuristic or algorithm which treats only a subset of the constraints of the problem (the "easy" constraints). If the probe produces a solution which also happens to satisfy the remaining "hard" constraints of the problem, search can stop. Otherwise, the solution provided by the probe is analyzed to find a violation of the "hard" constraints, and a variable involved in the conflict is chosen for branching. This method as described is not related to LS, but LS may be used in the probe. In [51], the authors concentrate on scheduling problems. A probe is implemented which respects only precedence constraints and the objective, while relaxing (ignoring) capacity constraints of resources. Resource capacities which are violated by the probe mean that too many tasks execute at the same time. For a time when resources are oversubscribed, a branch is created which orders a pair of tasks (via the addition of a new precedence constraint in the "easy" constraints), driving the search toward satisfaction of the capacity constraints.

Lever [57] studies a network routing problem and proposed a combination of tree search and LS for solving it. At a high level, the search is one based upon tree search with propagation which decides for each potential demand whether it will be routed or not. After each positive decision in the search tree, the demand is routed with those already present, which may result in capacity violations in the network. The infeasibility is repaired using an LS process. If the LS process does not succeed in legally placing the new demand in conjunction with those already placed within a certain limit, then the branch is pruned in the search tree, and the order is not placed. Thus, the method is not complete, but can use constraint propagation to determine that certain demands must be routed, for example. Lever demonstrates that his method compares well to probe-backtrack search [50] on this problem.

Crawford [22] uses an LS technique to identify hard-to-satisfy clauses in an SAT formulation and from that, variables which should be prioritized for early assignment inside a complete technique. Although the idea seems promising, results indicate that only a 10% reduction in the number of backtracks is obtained.

Mazure et al. [61] mostly follow the same idea as [22], the essential difference being that Mazure et al. run the LS process before each instantiation of a variable in the complete process. After each LS, the variable which was the most "unsatisfiable" (by appearing most in unsatisfied clauses) is chosen for branching. (The technique can also find solutions when the LS technique delivers an assignment which satisfies all constraints.) The authors compare their method to standard techniques (such as choosing a variable in the shortest clause) and demonstrate large speedups on some problems.

Benoist and Bourreau [7] describe "branch-and-move," a technique which uses LS to guide a traditional depth-first search process. The idea of branch and move is to perform LS on a tentative assignment to the unassigned variables, trying to reduce constraint "unhappiness." The unhappiness is defined for each constraint as distance measure of the tentative assignment from the closest support. The algorithm uses this information to generate branching decisions which concentrate on taking corrective action to please the most unhappy constraint. The authors apply their algorithm to a TV-break scheduling problem, and find that their methods outperform MIP, LS, and standard CP techniques.

6 Conclusion

We have described, through reference to work carried out over the last two decades, how CP and LS methods can be combined. While we could not be exhaustive (in particular, there is much research in the SAT community which could be influential, for instance, using LS to prove non-satisfiability of SAT instances [4, 27, 84]), we have tried to show how the domain has progressed in many different ways. That said, we believe that there is still much to be done in this promising area. We list some possibilities:

1. LNS has been an extremely effective technique, but to our knowledge, there is very little work using LNS as a move operator inside another meta-heuristic technique, such as tabu search ([76] uses simulated annealing). Likewise, we also know of no work trying to apply LNS to decision problems rather than optimization problems. Finally, it would be interesting to see if intensification techniques such as streamlining [42] could be used to better explore larger fragments.
2. LS for increasing filtering power [30, 96] is also a technique which deserves more attention. The idea of finding a witness which shows that extensions to a partial solution are fruitless is a powerful technique. Until now, these techniques have been applied at a *problem* level. Of interest would be to find out if such techniques can be applied to certain global constraints, or groups of constraints, allowing their inferences to be re-used in different problems.
3. As it maintains a complete assignment and an exact evaluation of this assignment, LS can be a good technique for finding out where the difficult part of a problem may lie. We believe that more work (in the spirit of [61]) deserves to be carried out to analyze the information derived from LS in order to perform better branching.

As the problems we face become larger and more challenging and application areas for optimization and automated reasoning widen, solving these problems will result in hybridization through necessity: no single domain has all the best tricks. So, we should look forward to more hybridization in the future, in which CP and LS will certainly play a big part.

References

1. Aarts E, Lenstra JK (eds) (1997) Local search in combinatorial optimization. Princeton University Press, Princeton
2. Adorf HM, Johnston MD (1990) A discrete stochastic neural network algorithm for constraint satisfaction problems. In: Proceedings of the international joint conference on neural networks
3. Applegate D, Cook W (1991) A computational study of the job-shop scheduling problem. ORSA J Comput 3(2):149–156
4. Audemard G, Simon L (2007) GUNSAT: a greedy local search for unsatisfiability. In: Proceedings of IJCAI-07, pp 2256–2261
5. De Backer B, Furnon V, Shaw P, Kilby P, Prosser P (2000) Solving vehicle routing problems using constraint programming and metaheuristics. J Heuristics 6(4):501–523
6. Beck JC, Perron L (2000) Discrepancy-bounded depth first search. In: Proceedings of CP-AI-OR 2000
7. Benoist T, Bourreau E (2003) Improving global constraints support by local search. In: The CP 2003 workshop on cooperative solvers in constraint programming
8. Bent R, Van Hentenryck P (2004) A two-stage hybrid local search for the vehicle routing problem with time windows. Transporatation Sci 38(4):515–530
9. Bent R, Van Hentenryck P (2006) A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. Comput Oper Res 33(4):875–893
10. Bent R, Van Hentenryck P (2007) Randomized adaptive spatial decoupling for large-scale vehicle routing with time windows. In: Proceedings of AAAI-07, pp 173–178
11. Brailsford SC, Hubbard PM, Smith B, Williams HP (1996) The progressive party problem: a difficult problem of combinatorial optimisation. Comput Oper Res 23:845–856
12. Brelaz D (1979) New methods to color the vertices of a graph. Commun ACM 22(4):251–256
13. Carchrae T, Beck JC (2005) Cost-based large neighborhood search. In: Workshop on the combination of metaheuristic and local search with constraint programming techniques
14. Carchrae T, Beck JC (2008) Principles for the design of large neighborhood search. J Math Model Algorithm 8(3):245–270
15. Caseau Y (2006) Combining constraint propagation and meta-heuristics for searching a maximum weight hamiltonian chain. Oper Res 40:77–95
16. Caseau Y, Laburthe F (1995) Disjuntive scheduling with task intervals. Technical Report 95–25, Laboratoire d'Informatique de l'École Normale Superieure, Departement de Mathématiques et d'Informatique
17. Caseau Y, Laburthe F, Le Pape C, Rottembourg B (2001) Combining local and global search in a constraint programming environment. Knowl Eng Rev 16:41–68
18. Caseau Y, Silverstein G, Laburthe F (2001) Learning hybrid algorithms for vehicle routing problems. Theor Pract Logic Program 1(6):779–806
19. Caseau Y, Laburthe F (1999) Heuristics for large constrained vehicle routing problems. J Heuristics 5:281–303
20. Chabrier A, Danna E, Le Pape C, Perron L (2004) Solving a network design problem. Ann Oper Res 130:217–239
21. Codognet P, Diaz D (2001) Yet another local search method for constraint solving. In: Proceedings of the international symposium on stochastic algorithms: foundations and applications, pp 73–90
22. Crawford JM (1993) Solving satisfiability problems using a combination of systematic and local search. In: Second DIMACS challenge: cliques, coloring, and satisfiability
23. Danna E, Perron L (2003) Structured vs. unstructured large neighborhood search: a case study on job-shop scheduling problems with earliness and tardiness costs. In: Proceedings of CP 2003, pp 817–821
24. Davenport A, Tsang E, Wang C, Zhu K (1994) GENET: a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In: Proceedings of AAAI-94

25. David P (1997) A constraint-based approach for examination timetabling using local repair techniques. In: Selected papers from the second international conference on practice and theory of automated timetabling, pp 169–186
26. Fahle T, Schamberger S, Sellmann M (2001) Symmetry breaking. In: Proceedings of CP 2001, pp 93–107
27. Fang H, Ruml W (2004) Complete local search for propositional satisfiability. In: Proceedings of AAAI-04, pp 161–166
28. Focacci F, Laburthe F, Lodi A (2001) Local search and constraint programming. In: Proceedings of the metaheuristics international conference (MIC '01), pp 451–454
29. Focacci F, Milano M (2001) Global cut framework for removing symmetries. In: Proceedings of CP 2001, pp 77–92
30. Focacci F, Shaw P (2002) Pruning sub-optimal search branches using local search. In: Proceedings of CP-AI-OR 2002, pp 181–189
31. Fukunaga AS, Korf RE (2005) Bin-completion algorithms for multicontainer packing and covering problems. In: Proceedings of IJCAI-05, pp 117–124
32. Galinier P, Hao J-K (2004) A general approach for constraint solving by local search. J Math Model Algorithm 3(1):78–88
33. Galinier P, Hertz A, Paroz S, Pesant G (2008) Using local search to speed up filtering algorithms for some NP-hard constraints. In: Proceedings of CP-AI-OR 2008, pp 298–302
34. Geelen PA (1992) Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proceedings of ECAI-92, pp 31–35
35. Gendreau M, Hertz A, Laporte G, Stan M (1998) A generalized insertion heuristic for the traveling salesman problem with time windows. Oper Res 46(3):330–335
36. Gent I, Hoos H, Rowley A, Smyth K (2003) Using stochastic local search to solve quantified boolean formulae. In: Proceedings of CP 2003, pp 348–362
37. Gent IP, MacIntyre E, Prosser P, Walsh T (1996) The constrainedness of search. In: Proceedings of AAAI-96, pp 246–252
38. Gent IP (2002) Arc consistency in SAT. In: Proceedings of ECAI – 02, pp 121–125
39. Ginsberg M, McAllester DA (1994) GSAT and dynamic backtracking. In: International conference on the principles of knowledge representation (KR94), pp 226–237
40. Glover F, Laguna M (1997) Tabu Search. Kluwer, Boston
41. Godard D, Laborie P, Nuijten W (2005) Randomized large neighborhood search for cumulative scheduling. In: Proceedings of ICAPS, pp 81–89
42. Gomes C, Sellmann M (2004) Streamlined constraint reasoning. In: Proceedings of CP 2004, pp 274–289
43. Gomes C, Selman B (1997) Algorithm portfolio design: theory vs. practice. In: Proceedings of UAI-97, pp 190–197
44. Harvey W, Ginsberg M (1995) Limited discrepancy search. In: Proceedings of IJCAI-95, pp 607–615
45. Hoos H, Tsang E (2006) Chapter 5. Local search methods. In: Handbook of constraint programming. Elsevier, Amsterdam
46. Huang J (2008) Universal booleanization of constraint models. In: Proceedings of CP 2008, pp 144–158
47. Joslin DE, Clements DP (1999) "Squeaky wheel" optimization. J Artif Intell Res 10:353–373
48. Bayardo RJ Jr, Schrag RC (1997) Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of AAAI-97, pp 203–208
49. Jussien N, Lhomme O (2002) Local search with constraint propagation and conflict-based heuristics. Artif Intell 139(1):21–45
50. Kamarainen O, El Sakkout H (2002) Local probing applied to network routing. In: Proceedings of CP-AI-OR 2004, pp 173–189
51. Kamarainen O, El Sakkout H (2002) Local probing applied to scheduling. In: Proceedings of CP 2002, pp 155–171
52. Kautz H, Horvitz E, Ruan Y, Gomes C, Selman B (2002) Dynamic restart policies. In: Proceedings of AAAI-02, pp 674–682

53. Kilby P, Prosser P, Shaw P (2000) A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. Constraints 5(4):389–414
54. Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proc Am Math Soc 7(1):48–50
55. Laborie P, Godard D (2007) Self-adapting large neighborhood search: application to single-mode scheduling problems. In: Proceedings of MISTA-07, pp 276–284
56. Langley P (1992) Systematic and nonsystematic search strategies. In: Proceedings of the first international conference on artificial intelligence planning systems, pp 145–152
57. Lever J (2005) A local search/constraint propagation hybrid for a network routing problem. Int J Artif Intell Tool 14(1):43–60
58. Lhomme O (2002) Amortized random backtracking. In: Proceedings of CP-AI-OR 2002, pp 21–32
59. Lin S (1965) Computer solutions of the traveling salesman problem. Bell Syst Tech J 44:2245–2269
60. Mautor T, Michelon P (1997) Mimausa : a new hybrid method combining exact solution and local search. In: Second international conference on meta-heuristics
61. Mazure B, Saïs L, Grégoire É (1998) Boosting complete techniques thanks to local search methods. Ann Math Artif Intell 22:319–331
62. Michel L, Van Hentenryck P (1999) Localizer: a modeling language for local search. INFORMS J Comput 11(1):1–14
63. Michel L, Van Hentenryck P (2000) Localizer. Constraints 5:43–84
64. Minton S, Johnston MD, Philips AB, Laird P (1992) Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. Artif Intell 58(1):161–205
65. Mladenovic N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24(11): 1097–1100
66. Morris P (1993) The breakout method for escaping from local minima. In: Proceedings of AAAI-93, pp 40–45
67. Nareyek A (2001) Using global constraints for local search. In: Constraint programming and large scale discrete optimization. DIMACS, Vol 57, pp 9–28
68. Palpant M, Artigues C, Michelon P (2004) LSSPER: Solving the resource-constrained project scheduling problem with large neighborhood search. Ann Oper Res 131:237–257
69. Perron L (2002) Parallel and random solving of a network design problem. In: AAAI workshop on probabilistic approaches in search, pp 35–39
70. Perron L (2003) Fast restart policies and large neighborhood search. In: Proceedings of CP-AI-OR 2003
71. Perron L, Shaw P (2004) Combing forces to solve the car sequencing problem. In: Proceedings of CP-AI-OR 2004, pp 225–239
72. Perron L, Shaw P, Furnon V (2004) Propagation guided large neighborhood search. In: Proceedings of CP 2004, pp 468–481
73. Pesant G, Gendreau M (1996) A view of local search in constraint programming. In: In Proceedings of CP '96, pp 353–366
74. Pesant G, Gendreau M (1999) A constraint programming framework for local search methods. J Heuristics 5(3):255–279
75. Pesant G, Gendreau M, Rousseau J-M (1997) GENIUS-CP: A generic single-vehicle routing algorithm. In: Proceedings of CP '97, pp 420–433
76. Pisinger D, Røpke S (2007) A general heuristic for vehicle routing problems. Comput Oper Res 34(8):2403–2435
77. Røpke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Sci 40:455–472
78. Røpke S, Pisinger D (2006) A unified heuristic for a large class of vehicle routing problems with backhauls. Eur J Oper Res 171(3):750–775
79. Prescott-Gagnon E, Desaulniers G, Rousseau L-M (1999) A branch-and-price-based large neighborhood search algorithm for the vehicle routing problem with time windows. Networks 54(4):190–204

80. Prestwich S (2000) A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In: Proceedings of CP 2000, pp 337–352
81. Prestwich S (2001) Local search and backtracking vs non-systematic backtracking. In: AAAI 2001 Fall symposium on using uncertainty within computation, pp 109–115
82. Prestwich S (2001) Trading completeness for scalability: hybrid search for cliques and rulers. In: Proceedings of CP-AI-OR 2001, pp 159–174
83. Prestwich S (2002) Combining the scalability of local search with the pruning techniques of systematic search. Ann Oper Res 115:51–72
84. Prestwich S, Lynce I (2006) Local search for unsatisfiability. In: Proceedings of SAT 2006, pp 283–296
85. Årgen M, Flener P, Pearson J (2005) Incremental algorithms for local search from existential second-order logic. In: Proceedings of CP 2005, pp 47–61
86. Årgen M, Flener P, Pearson J (2005) Set variables and local search. In: Proceedings of CP-AI-OR 2005, pp 19–33
87. Refalo P (2004) Impact-based search strategies for constraint programming. In: Proceedings of CP 2004, pp 557–571
88. Régin JC (1994) An filtering algorithm for constraints of difference in CSPs. In: Proceedsings of AAAI-94, pp 362–367
89. Richards ET, Richards B (1998) Non-systematic search and learning: an empirical study. In: Proceedings of CP-98, pp 370–384
90. Richter Y, Freund A, Naveh Y (2006) Generalizing AllDifferent: the SomeDifferent constraint. In: Proceedings of CP 2006, pp 468–483
91. Rousseau L-M, Gendreau M, Pesant G (2002) Using constraint-based operators to solve the vehicle routing problem with time windows. J Heuristics 8(1):43–58
92. Savelsbergh MWP (1985) Local search in routing problems with time windows. Ann Oper Res 4:285–305
93. Schaerf A (1997) Combining local search and look-ahead for scheduling and constraint satisfaction problems. In: Proceedings of IJCAI-97, pp 1254–1259
94. Schrimpf G, Schneider J, Stamm-Wilbrandt H, Dueck G (2000) Record breaking optimization results using the ruin and recreate principle. J Comput Phys 159:139–171
95. Sellmann M, Ansótegui C (2006) Disco-Novo-GoGo: integrating local search and complete search with restarts. In: Proceedings of AAAI-06, pp 1051–1056
96. Sellmann M, Harvey W (2002) Heuristic constraint propagation – using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem. In: Proceedings of CP-AI-OR 2002, pp 191–204
97. Selman B, Kautz HA, Cohen B (1994) Noise strategies for improving local search. In: Proceeedings of AAAI-94, pp 337–343
98. Selman B, Levesque H, Mitchell D (1992) A new method for solving hard satisfiability problems. In: Proceedings of AAAI-92, pp 440–446
99. Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Proceedings of CP '98, pp 417–431
100. Shaw P, De Backer B, Furnon V (2002) Improved local search for CP toolkits. Ann Oper Res 115:31–50
101. Solnon C (2002) Ants can solve constraint satisfaction problems. IEEE transactions on evolutionary computation 6(4):347–357
102. Stuckey PJ, De La Banda MG, Maher M, Marriott K, Slaney J, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: mapping solver independent models to efficient solutions. In: Proceedings of the 21st international conference on logic programming. Lecture notes in computer science, vol 3668 . Springer, Heidelberg, pp 9–13
103. Taillard E (2003) Heuristic methods for large centroid clustering problems. J Heuristics 9(1):51–73
104. Taillard ED, Voss S (2001) POPMUSIC: Partial optimization metaheuristic under special intensification conditions. In: Hansen P, Ribeiro C (eds) Essays and surveys in metaheuristics. Kluwer, pp 613–629

105. Tamura N, Banbara M (2008) Sugar: A CSP to SAT translator based on order encoding. In: Second international CSP solver competition, pp 65–69
106. Tamura N, Taga A, Kitagawa S, Banbara M (2006) Compiling finite linear CSP into SAT. In: Proceedings of CP 2006, pp 590–603
107. Van Hentenryck P, Michel L (2003) Control abstractions for local search. In: Proceedings of CP 2003, pp 65–80
108. Van Hentenryck P, Michel L (2005) Constraint-based local search. MIT Press, Cambridge
109. Van Hentenryck P, Michel L, Liu L (2004) Constraint-based combinators for local search. In: Proceedings of CP 2006, pp 47–61
110. Voudouris C, Tsang E (1999) Guided local search and its application to the travelling salesman problem. Eur J Oper Res 113:469–499
111. Walser JP (1997) Solving linear pseudo-boolean constraint problems with local search. In: Proceedings of AAAI-97, pp 269–274
112. Walsh T (2000) SAT v CSP. In: Proceedings of CP 2000, pp 441–456
113. Yagiura M, Ibaraki T, Glover F (2004) An ejection chain approach for the generalized assignment problem. INFORMS J Comput 16(2):133–151
114. Yokoo M (1994) Weak-commitment search for solving constraint satisfaction problems. In: Proceedings of AAAI-94, pp 313–318
115. Yugami N, Ohta Y, Hara H (1994) Improving repair-based constraint satisfaction methods by value propagation. In: Proceedings of AAAI-94, pp 344–349
116. Zhang J, Zhang H (1996) Combining local search and backtracking technqiues for constraint satisfaction. In: Proceedings of AAAI-96, pp 369–374