

Local Search and Constraint Programming¹

Filippo Focacci*, François Laburthe•, Andrea Lodi°

*ILOG S.A.,
9, rue de Verdun
BP 85, 94253 Gentilly, France
ffocacci@ilog.fr

•BOUYGUES - Direction des Technologies Nouvelles
1, avenue E. Freyssinet
78061 St-Quentin-en-Yvelines Cedex, France
flaburthe@bouygues.com

°DEIS, University of Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
alodi@deis.unibo.it

1 Introduction

Real-world combinatorial optimization problems have two main characteristics which makes them difficult: they are usually large (see, for example, [Caprara et al., 1997], which describes real-world crew scheduling applications), and they are not *pure*, i.e., they involve a heterogeneous set of side constraints (see, e.g., union contract regulations for crew scheduling and rostering again in [Caprara et al., 1997]).

Hence, in most cases, exact approaches cannot be applied to solve real-world problems, whereas incomplete methods, and among them *Local Search* (LS) ones, have been proved to obtain very good results in practice (see many examples throughout the book).

LS techniques are based on a simple and general idea. Let P be the combinatorial optimization problem we want to solve, and s a current solution which, for the moment, we assume to be feasible for P , and to have value $z(s)$. A *neighborhood* is defined for s with respect to a *move type* \mathcal{N} , i.e., a *function* mapping s (actually, any feasible solution of P) in a subset $\mathcal{N}(s)$ of the overall solution space. In other words, $\mathcal{N}(s)$ contains all the feasible solutions of P which can be reached from s by means of a move of type \mathcal{N} . Examples of moves are given throughout the paper, but, roughly speaking, the move is a manipulation of s whose effect is the transition to another solution $x \in \mathcal{N}(s)$. The LS framework explores the neighborhood by searching for the solution $x^* \in \mathcal{N}(s)$ such that $\delta z = z(s) - z(x^*)$ is maximized (for minimization problems). If $\delta z > 0$, then an improved solution has been found, and the process is iterated by considering x^* as new current solution. Otherwise, a local optimum has been reached, and several very effective techniques can be applied to escape from it (see almost all the chapters of this book).

If problem P presents relevant feasibility issues, i.e., it is not easy in practice to find a feasible initial solution, the same framework can be applied anyway by using as current solution an infeasible one. In order to drive the local search process towards feasible solutions, the cost function needs to be modified to measure the infeasibility of the candidate solution. In this sense, the evaluation of a move can be more complex (penalty functions are usually necessary), but the framework does not change substantially.

¹Notes for the lecture in CP-AI-OR'02, School on Optimization. To appear in F. Glover and G. Kochenberger (Eds.), *Handbook on Metaheuristics*, Kluwer Academic Publishers.

Three important issues must be taken into account when dealing with real-world problems.

1. Huge problems require large neighborhoods whose exploration can be computationally expensive.
2. When dealing with problems involving many heterogeneous side constraints it is often preferable to consider them as hard constraints rather than transforming them into penalty functions. In these cases, the neighborhood may contain few feasible solutions, and again large neighborhoods are required in order to avoid getting trapped into local minima too often.
3. Real-world applications typically lead to frequent update/addition of constraints (recall again union contract regulations), thus the algorithmic approach requires flexibility.

In the last decade Constraint Programming (CP) has shown its effectiveness in modeling and solving real-world combinatorial optimization problems. CP is a programming paradigm exploiting Constraint Satisfaction techniques ([Mackworth, 1977]), and in the following we restrict our attention to CP on *Finite Domains* (CP(FD)) which is the case of all constraint tools for discrete optimization such as CHIP ([Aggoun and Beldiceanu, 1992]), Ilog Solver ([Solver, 2000]), Eclipse ([Schimpf et al., 1997]), cc(FD) ([van Hentenryck et al., 1993]), CHOCO ([Laburthe, 2000]), etc.

The next paragraph briefly introduces the CP terminology that will be used in the remainder of the chapter. Complete definitions can be found in [Marriott and Stuckey, 1998] (a complete textbook), and [Focacci et al., 2000b] (a basic introduction).

Combinatorial optimization problems are modeled through CP(FD) by means of a set of variables taking their value on a finite domain of integers, and are linked by a set of constraints. The constraints can be of mathematical or symbolic type, and in this second case, when they refer to a set of variables, are referred to as *global constraints*.

Global constraints typically model a well-defined part of the overall problem, where well-defined means that the same part has been recognized as a subproblem in several cases. A classic example of global constraint is the *all_different*(x_1, \dots, x_n) constraint which imposes that variables x_1, \dots, x_n must assume different values in a feasible solution.

To each constraint is associated a *propagation* algorithm aimed at deleting from variable domains the values that cannot lead to feasible solutions. Constraints interact through shared variables, i.e., as soon as a constraint has been propagated (no more values can be eliminated), and at least a value has been eliminated from the domain of a variable, say v , then the propagation algorithms of all the other constraints involving v are triggered ([Mackworth, 1977]).

Propagation algorithms are usually incomplete: once propagation is finished, there may remain some inconsistent values in the variable domains². Therefore, unless the propagation phase ends with a fully instantiated solution or a failure (proving the problem inconsistent), a *search* phase is executed. One branching step is performed by partitioning the current problem (or the subproblem) into (easier) subproblems, e.g., by instantiating a variable to a feasible value in its domain. Propagation and search are interleaved in order to reach one or all feasible solutions.

As soon as a feasible solution improving the current best one is found, CP systems add a new constraint to the remaining search tree stating that further solutions should have a better value. This new constraint excludes leaf nodes from the remainder of the tree having a cost which is worse than the current one. Thus,

²Note that in the general case forcing arc consistency even for a single constraint is NP-hard.

CP solves a sequence of feasibility problems that improve the value of the objective function.

It is not difficult to see that the main advantage of using CP systems is flexibility: CP supports the design of declarative, compact and flexible models where the addition of new constraints is straightforward and does not affect the previous model. Indeed, the propagation of the previous constraints remain unchanged (since they locally model parts of the overall problem), and the previous constraints simply interact with the new ones through shared variables.

Thus many real-world combinatorial optimization problems may benefit from the efficiency of LS as well as from the flexibility of CP. Throughout this paper, we review hybrid methods that combine principles from both methods. A first set of such hybrids belongs to the family of local search methods (going from one solution to a neighbor one) and use CP as a way to efficiently explore large neighborhoods with side constraints. A second set belongs to the family of global search (tree search) methods and use LS as a way to improve some of the nodes of the search tree or to explore a set of paths close to the path selected by a greedy algorithm in the search tree. In short, LS may use ideas from CP in order to make large neighborhoods more tractable, while CP may use ideas from LS to explore a set of solutions close to the greedy path in a tree search and converge more quickly towards the optimum.

The chapter is organized as follows. In Section 2 we present the techniques to combine LS and CP within hybrid algorithms, and we briefly review the literature on the topic. In Section 3 we discuss in details a didactic case study by presenting examples of the techniques described in Section 2 with respect to this specific problem. Finally, some conclusions are drawn in Section 4

2 Basic techniques

2.1 Constrained Local Search

As mentioned in Section 1, hybrid approaches combining LS and CP are particularly suitable for real-world combinatorial problems which are typically huge in size and involve side constraints. In these cases both the size of the problems and the presence of side constraints lead looking for optimality out of practice. However, even standard LS algorithm can get into trouble from a computational point of view with these problems since the size of the neighborhood grows very fast and/or testing solution feasibility is expensive.

2.1.1 Small neighborhoods with side constraints

Fast LS algorithms typically uses neighborhoods of small size which can be explored with a relatively small computational effort. Classic examples are the neighborhoods defined by a move which simply exchange a pair of assignments of the current solution. This kind of move has a wide domain of application, it is referred as *2-opt*, and has been classically used by [Lin and Kernighan, 1973] for the *Traveling Salesman Problem* (TSP). In the TSP case, given a Hamiltonian cycle (current solution), i.e., a sequence of edges connecting the cities in their order of visit, the 2-opt move simply deletes two of these edges by replacing them with two others in order to obtain a new feasible cycle.

It is well-known that 2-opt can be implemented so as to require an overall time complexity of $O(n^2)$ to find the move maximizing the improvement, i.e., to find the solution whose overall length is minimal among all neighbors of the current cycle.

Even if the neighborhood is in principle quite small, the addition of side constraints can considerably increase the computational effort required to explore it

since it is often necessary to test feasibility.

A typical example is the time-constrained variant of TSP in which the salesman needs to visit the cities within specific *Time Windows* (TSPTW). In this case, in order to find the best 2-exchange move we need to test feasibility, which means testing, for each move, if the resulting solution violates some of the time window constraints. This is the standard way of addressing problems with side constraints in LS: the neighborhood of the pure problem is explored, and for each neighbor, the side constraints are iterated and, for each one of them, its satisfaction is tested. Thus, note that to each constraint must be associated an algorithm testing its satisfaction for a given solution. Note also that as soon as side constraints are added, the computational overhead of constraint checks for LS increases. However, checking feasibility only at the very end of the decision process is only a passive way of using constraints; constraints may be used in more active way by factoring out some of the constraint checks early on in the iteration. Therefore, single checks may discard (hopefully large) sets of neighbors, thus improving the overall efficiency of the neighborhood exploration.

In the example of the 2-opt neighborhood for the TSPTW, one check of time window constraints takes $O(n)$ time, therefore a straightforward implementation of 2-opt for the TSPTW takes $O(n^3)$ time. However, smart incremental computations can reduce the complexity of the TSPTW 2-opt to $O(n^2)$ by caching earliest arrival times and latest departure times ([Kindervater and Savelsbergh, 1997]). Such optimized neighborhood explorations and constraint checks require specialized code that must be substantially changed whenever new side constraints are considered.

The main point concerns, however, the effectiveness of small neighborhood for real-world applications. With the addition of side constraints the number of feasible solutions of the neighborhood becomes smaller, thus the local optimization process is more likely to get trapped into local optima. Therefore real-world problems require larger neighborhoods, and exploring them by simple enumeration becomes ineffective (the same holds since the size of the problems typically grows).

2.1.2 Exploring large neighborhood with CP

As neighborhoods grow larger, finding the best neighbor becomes an optimization problem on its own, thus the use of global search is preferable over blunt enumeration.

We review two possibilities for implementing a LS algorithm, using CP. In both cases, we suppose that we have at hand a current feasible solution s and a CP model of the problem.

The first method consists in keeping a fragment of the solution s (keeping the value assignment for a subset of the variables), erasing the value of all variables outside that fragment and solving the subproblem defined by the uninstantiated variables. This technique was introduced by [Applegate and Cook, 1991] for job-shop scheduling. The job sequence is kept on all machines but one, and the scheduling subproblem on that machine is solved to optimality by branch-and-bound. In this case, the fragment corresponds to the sequencing order on all machines but one. For scheduling, the approach was generalized to other fragments (time slice, ranks, sets of ordering decisions etc.) by [Caseau and Laburthe, 1996] and [Nuijten and Le Pape, 1998]. The same approach was also applied to quadratic assignment problems by [Mautor and Michelon, 1997], and to vehicle routing by [Shaw, 1998]. Such fragment-based LS methods are usually easy to implement once a first CP model has been built. A number of constraint based tools can be used to improve their efficiency:

- an optimization constraint can be set on the neighborhood imposing that only

neighbors that strictly improve the objective value over the current solution are generated (see, e.g., [Nuijten and Le Pape, 1998]);

- fragment based neighborhoods can be explored in a Variable Neighborhood Search (VNS, see, [Mladenović and Hansen, 1997] for its definition, and see, [Caseau and Laburthe, 1996] for its application to fragment-based LS);
- in order to speed up the procedure, each neighborhood defined by a fragment can be explored by an incomplete search, such as Limited Discrepancy Search (see, [Harvey and Ginsberg, 1995] and Section 2.2.5).

The second method, introduced in [Pesant and Gendreau, 1996] and [Pesant and Gendreau, 1999], consists in modeling the exploration of a neighborhood through CP variables and constraints. Roughly speaking, a CP model of the neighborhood is created such that every feasible solution of the CP problem represents a move that transforms the current solution into a neighbor solution. As an example one can consider the classical *swap* move that swaps the values of two variables x_i and x_j . The neighborhood associated to the move can obviously be explored by two nested loops over indices i and j . Alternatively, the neighborhood may be defined by a CP model with two domain variables I, J and one constraint $I < J$. Every feasible solution (i, j) of the problem defined by variables I, J and constraint $I < J$ uniquely identifies a swap move. With such a model, the exploration of the neighborhood by means of iterators (two nested loops) can be replaced by a tree search (such as branch-and-bound for finding the best move).

Formally, the neighborhood $\mathcal{N}(s)$ of a solution s is described by a CP model N_P such that there is a one-to-one mapping between the set of solutions of N_P and the set of neighbors $\mathcal{N}(s)$. We refer to N_P as the neighborhood model and to its decision variables as neighborhood variables. In the framework proposed by [Pesant and Gendreau, 1996, Pesant and Gendreau, 1999], local search is then described as a sequence of CP tree search on auxiliary problems N_P .

While searching for a neighbor, two CP models are active: the original model for P , and the neighborhood model for N_P . The two models communicate through *interface constraints* linking variables across P and N_P .

In the example given before, the interface constraints are:

$$\begin{aligned} x[I] &= s[J] \wedge x[J] = s[I] \\ \forall k \quad I \neq k \wedge J \neq k &\Rightarrow x[k] = s[k] \end{aligned}$$

In addition, a cost function for the neighborhood model N_P can be defined, and branch-and-bound search can be used on N_P to find the best neighbor. These CP models support fast neighborhood explorations. Indeed, constraints are used not only for testing the feasibility of solutions (neighbors) once they have been generated, but also for removing during the search, through propagation, sets of infeasible neighbors. For instance, the values of the already instantiated neighborhood variables may cause the reduction of the domains of the problem variables through the interface constraints and the domain reductions for the problem variables may, in turn, back-propagate on other not yet instantiated neighborhood variables, removing the possibility to generate infeasible neighbors. Propagation can also reduce the search space when only improving neighbors or only the best neighbors are looked for: the bounding constraint on the cost of the move can propagate out non optimal neighbors. Propagation is thus able to discard infeasible or uninteresting portions of the neighborhood without actually iterating those sets of neighbors. The larger and the more constrained the problem, the more significant the reduction in neighborhood search provided by propagation.

Several other advantages can be identified in such a CP approach. First, a clear separation between problem modeling and problem solving is maintained. Modeling constraints for P are kept separate from the neighborhood model. This supports, for example, the addition of side constraints to P without changing the neighborhood model nor the search methods. Second, any branching scheme may be used for building and exploring the neighborhood search tree. The simplest idea would only instantiate variables from N_P , but branching may also be performed on variables from P or on variables from both P and N_P . In addition, efficient exploration strategy like Limited Discrepancy Search may be used instead of Depth First Search. Few works have started taking advantage of this flexibility and the assessment of its interest is still an open research issue.

The main limitation of this approach lies in the overhead from the CP model and the propagation engine. CP models of the neighborhoods are of interest only when propagation produces a significant reduction of the search space; in such cases, the CP search of the neighborhood generates much fewer neighbors than the nested loop iteration. Moreover, the search tree exploration keeps instantiating and uninstantiating variables (upon backtracking). Searching the swap CP neighborhood with a simple branching scheme takes $O(n^3)$ time. Specific branching schemes have been proposed in [Shaw et al., 2000] to reduce this complexity to $O(n^2 \log n)$.

2.2 Incomplete Global Search

Local search and constraint propagation can also be applied within a global search algorithm. Focusing on the family of constructive algorithms, this section shows that, on the scale from greedy algorithms to complete global search, incorporating ideas from local moves and neighborhoods within global search is useful for achieving interesting compromises between solution quality and search time.

2.2.1 Constructive algorithms

A global search algorithm produces a solution by taking decisions and backtracking on failure. The decisions taken in a branch amount to adding a constraint to the problem. Some general branching scheme, such as the first-fail criterion (see, [Haralick and Elliott, 1980]) will select any variable from the model (that with the smallest number of values in its domain) and instantiate it: in such general cases, it is often difficult to interpret the state of the system before a solution has been reached. The situation is different for some branching schemes that are problem specific and where the decisions at each choice point build a small part of the final solution. For instance, in the case of vehicle routing, insertion algorithms consider customers one by one and decide the route that will visit them; for scheduling, ranking algorithms construct the schedule of a machine in a chronological manner by deciding which task should be sequenced first, which second, and so on; for time-tabling, assignment algorithms decide of the duty of a person (or a group of people) for one time-slot. Such global search algorithms are called *constructive search algorithms*: their states may indeed be interpreted as relevant partial solutions (routing plans for a subset of the customers, short-term schedules planning only a subset of the tasks or time-tables for a subset of the people) and it is easy to evaluate a bound of the objective function by adding the contribution from past decisions to an evaluation of the impact of the decisions to come.

2.2.2 Greedy constructive algorithms

The search in a constructive algorithm is guided by a heuristic³: at each choice point, a function h is evaluated for all possible choices and the choices are ranked by increasing values of h : the choice that minimizes h is considered the preferred decision. In a greedy constructive algorithm, the preferred branch is systematically followed and no backtracking takes place. For pure optimization problems where feasibility is not an issue, such greedy algorithms yield a solution in polynomial time. In case of feasibility issues, the algorithm may produce a partial solution (some customers are not assigned to a route, some tasks are not scheduled, some duties are not assigned in the time-table).

When the optimization system is granted more time than what is required by the greedy constructive algorithm, but not enough for performing a complete global search, local search may be an effective tool for improving the greedy solution. A first idea consists in using the greedy solution as starting point for a descent search or any random walk. However, interesting results can also be achieved by integrating notions from local search directly within the construction process.

The idea is that the construction process should explore a neighborhood of the greedy decision at each step of the construction process. Such a result can be reached in several ways: by performing a lookahead evaluation of the quality of branches (see Section 2.2.3), by considering a subset of the branches that are “close” to the best branch selected by the heuristic (see sections 2.2.4 and 2.2.5), or by trying to improve the current solution by a LS algorithm after each construction step (see Section 2.2.7).

2.2.3 Lookahead algorithms

Simple heuristics for evaluating the interest of a branch are often “myopic” in the sense that they only assess a choice by some of its immediate consequences and not by long-term consequences on the planning process. For instance, in vehicle routing one may evaluate the insertion of a client in a route by the minimal distance between the client and any other client already in the route. A possibility for taking into account such far-reach effect consists in going down the branch, fully propagating the effects of the choice and evaluating a heuristic only thereafter. In the example of vehicle routing, this amounts to performing the insertion of the client at the best place in the route, propagating the consequences of the insertion, and returning the bound of the overall cost. One may also perform a deeper exploration of the subtree below each branch before evaluating and ranking them. Such lookahead heuristics are inspired from game theory where the players may select their moves by unrolling the game n moves ahead and choosing the branch from which the worst reachable situation is best. In the field of combinatorial optimization, lookahead evaluation is a common way of improving greedy algorithms in vehicle routing (see, e.g., [Caseau and Laburthe, 1999]) or scheduling (see, e.g., [Dell’Amico and Trubian, 1993]).

2.2.4 Restricted Candidate Lists

At each choice point, the heuristic provides a preferred branch, as well as an indication of the quality of the other branches. In case of binary branching schemes, the heuristic may indicate how close both possibilities are, with situations ranging from near ties to definite choices between one good option and a terrible one. In the case of wider (non binary) branching, the heuristic may consider that some branches

³Note that, in the CP context, the word ‘heuristic’ does not refer to an approximation algorithm, but to a function used to compare different branches at a choice point. In the remainder of the chapter, heuristic will always refer to that meaning.

are serious competitors to the favorite branch while others are not. In any case, one can explore a subset of all solutions by following only those paths in the tree that never consider a poor branch according to the heuristic. Let b_1, \dots, b_k be the possible choices (branches), b_1 the preferred one ($h(b_1) = \min_i(h(b_i))$) and b_k the worst one ($h(b_k) = \max_i(h(b_i))$). The idea of Restricted Candidate Lists (RCL, see [Feo and Resende, 1995] and [Glover, 1995]) is to retain only the good branches and to discard the bad ones. More precisely, given a parameter α such that $\alpha \in [0, 1]$, only those b_i such that $h(b_i) \leq h(b_1) + \alpha(h(b_k) - h(b_1))$ are kept in a RCL while the others are discarded.

The introduction of RCL thus defines a subtree of the overall search tree. For $\alpha = 0$, this subtree covers all solutions reachable by a greedy algorithm; on the opposite $\alpha = 1$ yields a complete tree covering all possible solutions. For intermediate values of α , the subtree contains only solutions whose construction paths are located “around” greedy paths.

Such a subtree can be explored either systematically or not. In both cases it is important to control the global amount by which a solution path will diverge from the heuristic. It is indeed favorable to generate first solutions that diverge little from the heuristic (following good branches from the RCL) over solutions that systematically diverge from the heuristic (following always the worst branches from the RCL).

The GRASP method ([Feo and Resende, 1995]) uses RCL within a randomized version of the greedy algorithm: a randomized version of the overall construction algorithm is run many times. For each construction, at each choice point, one branch is selected at random among the RCL and according to some probability distribution (with decreasing probabilities for b_1, b_2, \dots, b_k). Thus, solutions that are globally closer to the heuristic are generated with an overall higher probability than solutions that are systematically far from the heuristic. Much room is left for tuning a GRASP algorithm, through the probability distributions or through the value of α . For instance, [Prais and Ribeiro, 1998] showed that it is more efficient to consider varying values of α , starting with tight ones (around 0, in order to follow the heuristic), and progressively releasing their values to accept locally bad choices (higher values). Another possibility consists in allowing higher values of α early on in the construction process (at the first levels of the tree) and restricting the construction to quasi-greedy choices with low values of α in the end (deep in the tree); this is motivated by the fact that heuristics are often more reliable in the end of the construction process rather than at the beginning.

2.2.5 Discrepancy-based Search

The previous section introduced the notion of a subtree defined by RCL for a given heuristic. This subtree can either be explored by means of randomized construction procedures (GRASP) or by systematic search. This is, in essence, what discrepancy-based search procedures (xDS, [Harvey, 1995]) do: this subtree is explored with construction moves and backtrack moves.

xDS add two notions to RCL:

- it keeps track of all nodes where the algorithm has diverged from the heuristic. Such cases when a branch b_i , $i > 1$ is followed are called discrepancies;
- it keeps track of the paths already generated in order to avoid visiting them twice and relying on backtracking to avoid recomputing many times common intermediate nodes.

This global account of the amount of discrepancy from the greedy heuristic is used to drive the exploration towards solutions that diverge little from the heuristic

(i.e., following most of the times the branch b_1) before solutions that diverge more from the heuristic (i.e., following many b_i branches with $i > 1$). Thus, xDS methods in a way ensure by explicit discrepancy bounding what GRASP ensures on average, through cumulated probabilities. The underlying principle is the same in both methods: it is assumed that good solutions are more likely to be constructed by following always but a few times the heuristic b_1 , rather than by diverging often from it.

The global account of discrepancies can be performed by several manners:

- counting the number of times K the search did not follow the heuristic. Limited Discrepancy Search (LDS, [Harvey and Ginsberg, 1995]) explores the tree by generating all solutions for increasing values of K ;
- counting the number of times the search did not follow the heuristic up to a certain depth. Depth-bounded Discrepancy Search (DDS, [Walsh, 1997]) explores the tree by generating all solutions that do not exceed a maximum number of discrepancies up to a certain depth, and then strictly follows the greedy algorithm;
- counting the total divergence in rank between the options taken and the preferred ones (with C denoting the sum of the rank discrepancy, i.e. $i - 1$, for all branches b_i that are followed along the path). For two consecutive choices, this method associates the same divergence to a the path taking b_1 and then, b_3 and to a path taking twice the decision b_2 . Credit Search, ([Beldiceanu et al., 1999]) generates all solutions for which C does not exceed a given limit.

2.2.6 Local search over priority list

In a similar spirit, in the case of priority-based heuristics, local moves can be applied directly to the priority list itself. Consider a problem P and a static heuristic for a constructive algorithm $L = (o_1, o_2, \dots, o_n)$, where L is a list of decision objects o_i of P (e.g. a list of variables in a generic CSP, a list of activities in a scheduling problem). A constructive algorithm A sequentially makes decision on o_i by adding a constraint involving o_i . If the algorithm A stops at the first solution found, then it can be seen as a function mapping the constructive heuristic L into a solution S of the problem P . Given this interpretation, we can apply local search methods to the static heuristic instead of applying them directly to the solution S (for example by considering all lists generated by exchanging each pair of decision objects o_i, o_j in L). Note a similar technique is sometimes used in Genetic Algorithms (GA). A GA engine may work on an indirect representation of the problem P (“genotype”), and use a function to map each genotype to a “phenotype” representing a solution of the real problem. Since most of the times the mapping between genotype and phenotype needs to consider complex constraints, and it may be implemented as a constraint programming engine. In this case the static heuristic is generated by the GA engine; more generally, a static heuristic may be generated by a different algorithm A_1 working on a problem P_1 related to P . Two different ways can be used to explore the neighborhood of a static heuristic L : the first method applies local search directly to the list L . The second method interprets the rank i of a decision object o_i as a preference, and explore an incomplete tree search (for example using limited discrepancy search) within the preference-based framework as in ([Junker, 2000]). After having found a first solution by following the given order, preference based search is able to explore permutation of such an order by backtracking on a tree search while looking for new solutions.

2.2.7 Improving solutions

A last possibility for enhancing an (incomplete) global search algorithm consists in applying some local search steps.

- Local search can be applied at a leaf of the global search tree for improving a solution. This is a straightforward generalization of local search methods which build a solution by a greedy algorithm. Global search is simply used as a way to generate several initial solutions on which a local search improvement phase is applied. It is interesting to generate starting solutions that are different enough for the overall exploration to be rather diversified. LDS is an interesting way of generating such a diversified initial set of solutions.
- Local search can be applied at internal nodes of the global search tree for repairing or improving a partial solution (see [Prestwich, 2000]). Designing such moves in the general case of any CP model may be hard. Indeed, such moves must handle partial assignments (producing a partial assignment similar to the current one). A simple idea of neighborhoods consists in selecting a set V_1 of variables that are instantiated in the current partial solution, produce a neighbor assignment of V_1 , and apply a propagation algorithm on the overall problem to reduce the domains of variables not in V_1 . The global search process can then continue from this improved partial state. Such methods have proven successful on routing problems. [Russell, 1995] introduced the idea of applying local moves every t steps of insertion: each local move tries to improve the partial plan (routes visiting a subset of the clients) by another partial plan, visiting the same clients, but in a different order, before continuing the insertion process. [Caseau and Laburthe, 1999] compared the method that applies LS after each insertion step (Incremental Local Optimization, ILO, see also [Gendreau et al., 1992]) to the method that constructs a solution by greedy insertion and then improves it by LS. They showed that ILO was not only faster but also produced much better solutions.

A key element to be considered with LS is the evaluation of partial solutions. The quality of a move can easily be measured when LS is applied on completely instantiated solutions. On the other hand, when LS is applied on partial solutions, evaluating a move may be more difficult. It is usually interesting to consider bounds on the objective function, possibly with the addition of a term evaluating the difficulty of extending the partial solution into a complete one.

2.3 Other related methods

A variety of other methods have been proposed in the last decade to solve combinatorial problems with constraints using local search, focusing on solving over-constrained problems (problems where one wants to minimize a global account of penalties for violated constraints). For problems that are described as generic constraint satisfaction problems, the GSAT by [Selman et al., 1992] and the Min-Conflict by [Minton et al., 1992] methods start from a random infeasible assignment of values to variables and improve it by flipping the value of one variable that is involved in the biggest number of conflicts (violated constraints). Improvements have been proposed over this original framework, with the careful introduction of some randomization in the choice of the variable to be flipped, leading to the walkSat algorithm by [Selman and Kautz, 1993]. However, such methods are based on very simple models of the optimization problems, with constraints that are described by the list of their feasible assignments. There is thus no means of capturing the entire structure of the problem by such methods. The walkSAT algorithm has recently

been generalized into the WSAT algorithm, capable of handling linear constraints over integer variables (see, [Walser, 1999]); however, as a generic method, it is unable to take advantage of the wealth of knowledge that has been developed for computing bounds over the cost of specific routing or assignment problems.

A completely different approach is followed by the localizer framework by [Michel and van Hentenryck, 1997]. Problems are described by means of variables and formulas called invariants. Although it seems similar to a modeling language, the invariants do not specify feasible solutions, but are rather used to define the way in which data structures are updated when a move is performed. The approach is powered by powerful symbolic reasoning over the invariants and has proven efficient on some problems. However, it does not yet support global objects for modelling a route, a schedule or an assignment.

Thus, for optimization problems featuring a strong combinatorial structure as well as specific side constraints one is left with the need to craft a specific method from the model. For this reason, we will not discuss these methods any further.

3 Practical Guidelines through a case study

The techniques for combining Local Search and Constraint Programming described in the previous section are applied here to a didactic case study: we address a variant of the classical *Vehicle Routing Problem* (VRP), (see, [Toth and Vigo, 2002]) in which several side constraints are considered modeling real-world requirements.

Informally, we are given a set of *clients* and a depot in which a fixed number of *trucks* are located. Each client produces a given amount of *goods* of a certain type and has to be visited within a *time window*. Early arrivals are allowed, in the sense that a truck can arrive before the time window lower bound, but, in this case, it has to wait until the client is ready for the beginning of the service. The service time, for each client, only depends on the type and quantity of goods to be collected (i.e., it does not depend on the truck), and each truck has two *capacitated bins* which each can contain a unique type of goods. The travel times and costs between each pair of clients, and between each client and the depot, are given.

This VRP variant, referred to as *didactic* Transportation Problem (dTTP) in the following, calls for the determination of the set of trucks' routes minimizing the total travel cost, and satisfying the following constraints:

- each route is associated to a truck, and starts and ends at the depot;
- each client is visited exactly once, and within the time window;
- the bins' capacity constraints are respected.

3.1 A CP model

The CP model of the transportation problem dTTP can be formally stated by using the following notation:

- $i, j \in \{1, \dots, N\}$ for the locations (0 denotes the depot) and, by extension, for the clients;
- $k \in \{1, \dots, M\}$ for the trucks, and, by extension, for their routes;
- $h \in \{1, \dots, 2M\}$ for the bins;
- $\ell \in \{1, \dots, P\}$ for the types of goods.

For each location i ($i = 1, \dots, N$) the corresponding goods are denoted by $type_i$ and q_i , where $type_i \in \{1, \dots, P\}$ indicates the type of goods to be collected by client i , while $q_i > 0$ is its quantity. Moreover, each client i has an associated time window $[a_i, b_i]$ representing the time frame during which the service must start. The fleet of vehicles located at the depot 0 is composed by M trucks: each truck k has two bins of identical capacity C , and each bin may collect a unique type of goods $\ell \in \{1, \dots, P\}$. The duration of the service at location i is $d_i > 0$. Finally, for each pair of locations i and j , the travel time $tt_{ij} \geq 0$ and the travel cost c_{ij} is reported. A possible CP model for dTP is the following:

$$\begin{aligned}
\min \quad & totCost = \sum_{k=1}^M cost_k \\
\text{on} \quad & \\
\forall k \in \{1, \dots, M\} \quad & cost_k \geq 0, \\
& truck_k = UnaryResource(tt, c, cost_k) \\
\forall h \in \{1, \dots, 2M\} \quad & collects_h \in [1..P] \\
\forall i \in \{1, \dots, N\} \quad & start_i \in [a_i..b_i], \\
& service_i = Activity(start_i, d_i, i), \\
& visitedBy_i \in [1..M], \\
& collectedIn_i \in [1..2M] \\
\text{subject to} \quad & \\
\forall i \in \{1, \dots, N\} \quad & service_i \text{ requires } truck[visitedBy_i] \quad (1) \\
\forall h \in \{1, \dots, 2M\} \quad & \sum_{i \mid collectedIn_i=h} q_i \leq C \quad (2) \\
\forall i \in \{1, \dots, N\} \quad & collects[collectedIn_i] = type_i \quad (3) \\
\forall i \in \{1, \dots, N\} \quad & visitedBy_i = \left\lceil \frac{collectedIn_i}{2} \right\rceil \quad (4)
\end{aligned}$$

CP models for combinatorial optimization problems consist in the definition of three different sets of objects: an objective function, decision objects, and constraints. In the example, we explicitly separate the three sets with the keywords *min*, *on*, and *subject to*. Moreover, most Constraint Programming languages provide modeling objects such as *Activities*, *Resources*, etc. Using such objects rather than only variables yields more concise models. Even if the actual syntax of such objects may vary depending on the specific CP language at hand, we may safely assume that the model proposed can easily be coded using most CP languages.

3.1.1 Basic model

We are given M trucks, $2M$ bins, and N clients that must be visited within a time window by exactly one truck. Each truck is a *UnaryResource* object containing the information on the travel time and cost among locations (clients), and a variable representing the total travel cost for the truck. The service at each client is an *Activity* object defined by a variable start-time, a constant duration, and a location. Constraint (1) ‘*service_i requires truck[visitedBy_i]*’ enforces that from $start_i$ to $start_i + d_i$ the $truck_k$ *UnaryResource* is used by the *Activity* $service_i$ without interruption. A *UnaryResource* cannot be used simultaneously by more than one *Activity*, and, in addition, it is not used during the time needed to move from location to location. Moreover a given time and cost must be considered before the first and after the last activities are executed. In case tt satisfies the *triangle inequality* ($tt_{i_1, i_3} \leq tt_{i_1, i_2} + tt_{i_2, i_3}$), an equivalent model is that for every pair of *Activity* $service_i$, $service_j$ such

that ‘ $service_i$ requires $truck_k$ ’, and ‘ $service_j$ requires $truck_k$ ’, $(start_i \geq start_j + d_j + tt_{ji}) \vee (start_j \geq start_i + d_i + tt_{ij})$. Note that the same objects used to model trucks and visits in dTP could also be used to model machines with maximal capacity equal to 1, and tasks with sequence dependent setup times and cost in scheduling problems. A client $service_i$ needs to be visited by exactly one of the M trucks, say the k -th, thus we model the alternative choice of trucks by defining, for each client, a variable $visitedBy_i$ referring to that index k . The requirement constraint is thus stated on the $truck_k$ array of alternative resources indexed by the variable $visitedBy_i$.

Each client i produces a quantity q_i of goods of type $type_i$. The variable $collectedIn_i$ identifies the bin used to serve client i . The bin capacity constraint (2) for each bin h simply states that the sum of all quantities q_i such that q_i is collected in h is less or equal to the maximal capacity C . The variable $collects_h$ identifies the type associated to bin h . The constraint requiring that each bin must contain goods of the same type can be stated using variables $collects_h$ and $collectedIn_i$ by imposing that for each client i the type associated to the bin used by the client i is equal to $type_i$ (3).

Bins with index $2k - 1$ and $2k$ are associated to truck k (bins 1 and 2 are placed on truck 1, bins 3 and 4 on truck 2, etc.). The link between bins and trucks is modeled with constraint (4).

The *basic model* correctly models dTP. Decision variables are start-time ($start_i$) and bin selection ($collectedIn_i$) variables for each client i . Once all variables $start_i$ and $collectedIn_i$ are instantiated to values satisfying all the constraints of the *basic model*, a solution is reached, since all the other variables of the model ($cost_k$, $visitedBy_i$, etc.) are instantiated by propagation.

Nevertheless, when dealing with routing problems, it may be convenient to explicitly manipulate the sequence of services performed by each truck. For this purpose an extension of the model is considered. This model, based on the abstraction of multiple path, is redundant with respect to the *basic model*, since it is not necessary for correctly modeling dTP.

3.2 Propagation

CP constraints embed domain propagation algorithms aimed at removing values from variable domains that are proven infeasible with respect to the constraint itself (see, [Mackworth, 1977]). Several propagation algorithms can be designed for the same CP constraint; we briefly sketch some algorithms that can be used to implement the propagation of the constraints used in the dTP model.

Concerning notation, given a variable v , in the following we will refer to the minimum (resp. maximum) value in its domain as $\mathbf{inf}(v)$ (resp. $\mathbf{sup}(v)$). Moreover, the domain itself is referred to as $\mathbf{domain}(v)$, whereas once v has been instantiated its value is returned as $\mathbf{value}(v)$.

3.2.1 Disjunctive Relations

Consider the *UnaryResource* object used to model the trucks and the *require* constraints linking objects *Activity* and *UnaryResource*. As mentioned these constraints state that two activities must not be executed simultaneously on the same unary resource, and a transition time must be granted between two subsequent activities. A simple propagation algorithm updates the start-time variable of activities by looking at all pairs of activities performed on the same resource and deducing precedence constraints. Consider two activities $service_i$, $service_j$, if $(\mathbf{inf}(start_j) + d_j + tt_{ji} > \mathbf{sup}(start_i)) \wedge (\mathbf{value}(visitedBy_i) = \mathbf{value}(visitedBy_j))$ then it can immediately be deduced that $service_i$ must precede $service_j$, i.e.,

$\mathbf{inf}(start_j) := \max\{\mathbf{inf}(start_j), \mathbf{inf}(start_i) + d_i + tt_{ij}\}$

and

$\mathbf{sup}(start_i) := \min\{\mathbf{sup}(start_i) + d_i, \mathbf{sup}(start_j) - tt_{ij}\} - d_i.$

This propagation rule finds new time bounds for the activities by considering the current time bounds, the capacity availability, and the resource assignments. Similarly, new possible assignments are deduced by considering current time bounds, and capacity available: if $(\mathbf{inf}(start_j) + d_j + tt_{ji} > \mathbf{sup}(start_i)) \wedge (\mathbf{inf}(start_i) + d_i + tt_{ij} > \mathbf{sup}(start_j)) \wedge (\mathbf{value}(visitedBy_i) = k)$ then $visitedBy_j \neq k$. More sophisticated propagation algorithms have been implemented in different CP languages, see for example Edge Finding techniques for unary and discrete resources, and constructive disjunctive techniques for alternative resources (see, [Nuijten, 1994]). These techniques have been successfully applied to Scheduling Problems and TSPTW (see, [Pesant et al., 1998]). In the proposed model, beside the modeling of capacity availability and transition times, the object *UnaryResource* is also responsible for maintaining a cost variable corresponding to the resource specific component of the total cost.

3.2.2 Linking trucks and bins

For its simplicity, it is worth describing in detail the constraint linking the variables used for the choice of trucks and bins, $visitedBy_i = \lceil collectedIn_i/2 \rceil$. The constraint is considered consistent iff the following condition hold:

$\forall k \in \mathbf{domain}(visitedBy_i) \exists h \in \mathbf{domain}(collectedIn_i) | k = \lceil h/2 \rceil$

and vice-versa

$\forall h \in \mathbf{domain}(collectedIn_i) \exists k \in \mathbf{domain}(visitedBy_i) | k = \lceil h/2 \rceil.$

A simple propagation algorithm iterates on all values k belonging to the domain of $visitedBy_i$ and checks for the existence of a value h in the domain of $collectedIn_i$ that satisfies the relation $k = \lceil h/2 \rceil$. If no value h is found, k is removed from the domain of variable $visitedBy_i$. The propagation algorithm also iterates on all values h belonging to the domain of $collectedIn_i$ and checks for the existence of a value k in the domain of $visitedBy_i$ that satisfy the relation $k = \lceil h/2 \rceil$. If no value k is found, h is removed from the domain of variable $collectedIn_i$.

3.2.3 Propagating costs

In many cases the calculation of good lower bounds on the objective function variable is especially important. As shown in [Focacci et al., 1999a] good lower bounds on the optimal cost can be used for discarding a priori uninteresting parts of the solution space. Moreover, lower bounds can also be used in greedy algorithms as described in Section 3.4.2 for guiding towards more promising parts of the solution space. In CP the link between the objective variable and the decision variables is maintained through a constraint.

In summary, three things should be stressed. First, whenever a simple or global constraint acts on several variables the modification of any of the variables triggers propagation algorithms that propagate the modification on the other variables involved. Second, for a given constraint several propagation algorithms can enforce different degrees of consistency. Finally, even without giving complexity results of the propagation algorithms, it is clear that some of them can be computationally more expensive than others. For example, checking a constraint for the feasibility on a set of instantiated variables is usually much easier than eliminating all values that can be proved to be infeasible on a set of yet uninstantiated ones. Therefore, the propagation algorithms to use may vary depending on the size of the problem, and on the type of solving procedure used.

3.3 Redundant routing model

Starting from the basic model of Section 3.1.1, a redundant model can be devised. The main advantage of using redundant models in CP consists in better filtering out inconsistent values. In this specific case, another important advantage is that many neighborhood structures can be easily defined by means of the additional variables ($next_i$, $succ_i$) introduced by the redundant models. Nevertheless, redundant models should be used with care. Depending on the problem size, the use of several linked models could eventually be computationally penalizing. In order to show a wide range of CP-based local search techniques, we used all redundant models simultaneously. Performance issues arising with very large instances could be addressed by using lighter models or other techniques such as decomposition.

The redundant *routing* model is the following:

$$\begin{aligned} \forall k \in \{1, \dots, M\} \quad first_k &\in [1..N] \\ \forall i \in \{1, \dots, N\} \quad next_i &\in [1..N + M], \\ &succ_i \in [\{\}.. \{1, \dots, N\}] \end{aligned}$$

$$multiPath(first, next, succ, visitedBy) \tag{5}$$

$$costPaths(first, next, succ, c, totCost) \tag{6}$$

$$\begin{aligned} \forall i, j \in \{1, \dots, N\} \quad j \in succ_i &\Leftrightarrow \\ (visitedBy_i = visitedBy_j \wedge start_j > start_i) & \tag{7} \end{aligned}$$

Let $G = (V, A)$ be digraph, and a partition of V be defined by a set S of M *start* nodes, a set I of N *internal* nodes, and a set E of M *end* nodes. A multiple path constraint *multiPath* enforces that M paths will exist starting from a start node, ending in an end node and covering all internal nodes exactly once. All internal nodes have one direct predecessor (previous node) one direct successor (next node).

Internal nodes are labeled $1, 2, \dots, N$, end nodes are labeled $N + 1, N + 2, \dots, N + M$, while start nodes are labeled $N + M + 1, N + M + 2, \dots, N + 2M$. The multiple path constraint makes use of four arrays of variables. Variable $first_k$ is associated with the start node $N + M + k$, and identifies its next node in the path. Variables $next_i$, $succ_i$, and $visitedBy_i$ are associated with the internal node i . Variable $next_i$ identifies the node next of i ; variable $succ_i$ identifies the set of internal nodes occurring after i in the path containing i ; finally, variable $visitedBy_i$ identifies the path containing node i .

The *multiPath* constraint can be used to model a subproblem of dTP: each internal node represents a client, the set of start and end nodes represents copies of the depot. The path k starting from the start node $N + M + k$, covering a set of internal nodes, and ending at an end node represents the route of *truck_k*. The *redundant routing model* and the *basic model* are linked by the variables $visitedBy_i$ used in both models, and by the constraint (7).

Note that the cost on each truck only depends on the sequence of services performed, therefore the multiple path structure can also be used to define the objective function. Since the variable $next_i$, and $succ_i$ will be extensively used as decision variables for dTP, it is indeed useful to explicitly link the multiple path model to the cost variables $cost_k$, as done with the constraint (6).

3.3.1 Propagation

Since the multi path model will be extensively used in the discussion of the local search methods, it is important to describe it in details. A model equivalent to the *multiPath* constraint based on only simple arithmetical and logical constraints is the following:

$$\forall i, j \in \{1, \dots, N\}, i > j \quad next_i \neq next_j \quad (8)$$

$$\forall k_1, k_2 \in \{1, \dots, M\}, k_1 > k_2 \quad first_{k_1} \neq first_{k_2} \quad (9)$$

$$\forall i \in \{1, \dots, N\}, \forall k \in \{1, \dots, M\} \quad first_k \neq next_i \quad (10)$$

$$\forall i, j \in \{1, \dots, N\} \quad next_i = j \Rightarrow succ_i = \{j\} \cup succ_j \quad (11)$$

$$\forall i, j \in \{1, \dots, N\} \quad j \in succ_i \Rightarrow visitedBy_i = visitedBy_j \quad (12)$$

$$\forall k \in \{1, \dots, M\}, \forall i \in \{1, \dots, N\} \quad first_k = i \Rightarrow visitedBy_i = k \quad (13)$$

$$\forall i \in \{1, \dots, N\} \quad i \notin succ_i \quad (14)$$

As mentioned in the model description, the set variables $succ_i$ are used to enforce precedence relations among nodes. An integer set variable is a variable whose domain contains sets of integer values. In analogy with the definition of lower and upper bounds on integer variables, lower and upper bounds can also be defined for an integer set variable: the lower bound, called *required set*, is the intersection of all sets belonging to the domain of the variable; the upper bound, called *possible set*, is the union of all sets belonging to the domain of the variable. The variable is considered instantiated when the domain contains a single set of integer values, thus the required set is equal to the possible set. Required set, possible set, and value of the set variable v will be referred to as $\mathbf{req}(v)$, $\mathbf{poss}(v)$, and $\mathbf{value}(v)$.

Constraints (8)-(10) force all internal nodes $\{1, \dots, N\}$ to have exactly one incoming, and one outgoing arc, while all start nodes are forced to have exactly one outgoing arc belonging to the set of internal nodes. Constraints (11) and (12) link $succ$ and $next$, and $succ$ and $visitedBy$ variables; constraint (11) links the $visitedBy$ variable of the first internal node in a path starting from node $N + M + k$ to the value k . Finally the last constraint prevents cycles in the graph.

The global constraint *multiPath* could indeed be written using all the mentioned constraints. However, more effective and efficient propagation algorithms can be designed enforcing exactly the same semantic. Basic propagation on the $next$ variables enforces that, whenever a variable $next_i$ is instantiated to a value t , the value t is removed from the domain of all variables $next_j, j \neq i$. A more powerful propagation can be obtained by using flow algorithms (see, [Régin, 1994]). Cycles can be avoided either by simple cycle removal algorithms, or by performing a more sophisticated propagation using strong connected components or isthmus detection. Some of the propagation that can be performed on the variables $succ_i$ exploit the transitivity of the $succ$ array: $i_2 \in \mathbf{req}(succ_{i_1}) \wedge i_3 \in \mathbf{req}(succ_{i_2}) \Rightarrow \mathbf{req}(succ_{i_1}) := \mathbf{req}(succ_{i_1}) \cup \{i_3\}$. This propagation rule can be read as follows: if i_2 is necessarily after i_1 , and i_3 is necessarily after i_2 , then i_3 is necessarily after i_1 .

Beside the simple link between $next$ and $succ$ variables in (11), more effective propagation can be performed. For example, if given two nodes i and j , the set of nodes that necessarily follows i has a non-empty intersection with the set of nodes that necessarily precedes j , then at least a node must exist in any path from i to j , therefore j cannot be the next of i . More formally, $\mathbf{req}(succ_i) \cap (\{1, \dots, N\} \setminus \mathbf{poss}(succ_j)) \neq \emptyset \Rightarrow next_i \neq j$. Note that this last propagation rule is not part of the arithmetic model of the *multiPath* constraint (8)-(14); this is a simple example of the extra propagation that can be done by a global constraint.

3.4 Constructive Algorithms

The model presented above could be solved as such by complete global search. However, such a solution is impractical in terms of computing time for realistic

instances (recall that the maximal size of plain VRPs that can currently be solved to optimality within minutes ranges from a few tens of clients for branch-and-bound methods to 80 for branch-and-cut methods, see [Toth and Vigo, 2002]). This section discusses constructive methods which are realistic for solving dTP instances with hundreds of clients.

3.4.1 Insertion algorithms

In the dTP model, when all CP variables in the arrays *visitedBy* and *succ* are instantiated, then the solution is fully known. Indeed, variables arrays *first*, *next* and *cost* can be instantiated from *succ*. When all these variables are instantiated, the objective function is instantiated although the *collectedIn* and *start* variables may be left with an interval domain. A simple greedy algorithm can instantiate all these remaining variables. For example one could iteratively choose each client *i* and instantiate *collectedIn_i* and *start_i* to its lower bound⁴.

We propose a constructive algorithm based on this dominance property using (*visitedBy_i*) and (*succ_i*) as CP decision variables for dTP. Clients are considered one after the other: for each client *i*, the algorithm instantiates the variable *visitedBy_i*, and for all *j* such that *visitedBy_j* has already been instantiated to the same value as *visitedBy_i*, the *succ* variables are reduced by enforcing one of the two decisions $j \in succ_i$ and $j \notin succ_i$.

Such an instantiation scheme implements a constraint-based version of the standard insertion heuristics for vehicle routing (see, [Golden and Assad, 1988]). Indeed, each time a client *i* is assigned to a truck, the ordering between *i* and all others clients assigned to that same route is settled, yielding a total order on all clients of the route. These ordering decisions implement the insertion of *i* between two consecutive clients in a route. Compared to standard insertion algorithms in Operations Research textbooks, this CP description adds two notions.

- The routing problem can be enriched with side constraints, such as time windows, bin assignments and bin capacity. When selecting the best possibility for insertion, feasibility is ensured. Rather than performing the insertion and checking for the satisfaction of all constraints, we rely on propagation to discard some of the infeasible assignments. Infeasible route assignments are removed from the domain of *visitedBy_i*, and infeasible orderings are discarded by reducing the domain of *succ_i*.
- The overall insertion process is seen as a monotonic process where variable domains are reduced. Indeed, the *next_i* variables may remain uninstantiated until the very end of the construction process (until the routes are full). Until then, the routes are only partially sequenced with precedence decisions: the relative order among clients already assigned to a given route is settled ($\forall i, j, i \in succ_j \vee j \in succ_i$), but there is room to insert new clients between such *i* and *j*. This description of insertion within “open” routes supports the evaluation of lower bounds on the route costs: at any point in the algorithm, each route *k* has been assigned a subset of clients i_1, \dots, i_p ($visitedBy_{i_1} = visitedBy_{i_2} = \dots = visitedBy_{i_p} = k$) which have been ordered: $i_2 \in succ_{i_1}, \dots, i_p \in succ_{i_{p-1}}$. When the costs c_{ij} satisfy the triangle inequality, the cost of the route $cost_k$ can be bounded with $cost_k \geq c_{0,i_1} + c_{i_1,i_2} + \dots + c_{i_{n-1},i_n} + c_{i_n,0}$.

The general framework for construction algorithms can thus be described formally as follows:

⁴This is true with the current set of constraints while one has to be careful in the case of addition of other constraints.

```

algorithm: INSERTION
for all clients  $i$ 
  for each possible value  $k$  in  $\text{domain}(\text{visitedBy}_i)$ 
    branch on
      try the assignment  $\text{visitedBy}_i = k$ 
      for all  $j$  such that  $\text{domain}(\text{visitedBy}_j) = \{k\}$ 
        branch on try  $j \in \text{succ}_i$ 
          or try  $j \notin \text{succ}_i$ 
      if some feasible insertions were found
        commit to one among them
      else return(failure)
return(success)

```

In the end of the algorithm, either a feasible routing plan that services all clients (success) or no solution (failure) is obtained⁵.

3.4.2 Greedy insertion

This general scheme can be refined into a greedy algorithm.

- In order to help the algorithm finding a routing plan covering all clients, it is wise to consider first the clients that are “difficult” to insert into a route (those for which the insertion into a partial route is most likely to be infeasible). We therefore sort all clients and use as priority criterion a function that returns higher values for clients far away from routes already built and with tight time windows over clients close to current routes and with a loose time window. The latter have indeed little propagation impact when assigned to a route while the former drastically augment the lower bound on the cost of the route and significantly reduce the possibilities of further assignments through time window constraints. An example of such a static criterion returning high values for nodes that should be considered first would be $\text{priority}(i) = tt_{0,i}/(b_i - a_i)$.
- Since all trucks are similar, each assignment of a client to a (still) empty route amounts to a route creation and is strictly equivalent. Therefore, when there exist several k such that for no client j , $\text{domain}(\text{visitedBy}_j) = \{k\}$, then, the tentative assignment ($\text{visitedBy}_i = k$) is considered for only one k .
- When several possible insertions are found to be feasible for a client i , rather than selecting any of them, we can choose the assignment $\text{visitedBy}_i = k$ and the ordering (reductions on succ_i) that causes the lower bound of the overall objective z to increase the least: this turns the constructive method into a greedy algorithm.

The greedy algorithm is then as follows:

```

procedure: BEST_INS( $i, k$ )
oldInf := inf( $\text{cost}_k$ )
try  $\text{visitedBy}_i = k$ 
for all  $j$  such that  $\text{domain}(\text{visitedBy}_j) = \{k\}$ 
  branch on try  $j \in \text{succ}_i$ 
    or try  $j \notin \text{succ}_i$ 
over all solutions, minimize  $\Delta := \text{inf}(\text{cost}_k) - \text{oldInf}$ 

```

```

algorithm: GREEDY CONSTRUCTION
sort all clients  $i$  by decreasing values of  $\text{priority}(i)$ 

```

⁵In case of failure, it is always possible to remove from the model those clients which could not be inserted by the solver and return a feasible routing plan covering only a subset of the clients.

```

for all clients  $i$ 
  for  $k \in \text{domain}(\text{visitedBy}_i) \mid \exists j \neq i, \text{domain}(\text{visitedBy}_j) = \{k\}$ 
    try BEST_INS( $i, k$ )
  if  $\exists k_0 \in \text{domain}(\text{visitedBy}_i) \mid \forall j \neq i, \text{domain}(\text{visitedBy}_j) \neq \{k_0\}$ 
    try BEST_INS( $i, k_0$ )
  if some feasible insertion was found
    commit to the one that yielded the smallest  $\Delta$ 
  otherwise return(failure)
return(success)

```

Adapting the greedy algorithm to a dynamic priority criterion is straightforward.

3.4.3 Discrepancy-based search

As any greedy heuristic, the previous algorithm can be transformed into a search algorithm through the addition of *backtracking*. By adding a limited amount of backtracking, the algorithm explores a more substantial portion of the solution space at the price of an increased complexity. Discrepancy-based Search (see, [Harvey, 1995]) produces solutions by following the preferred branch of the heuristics always but a limited number of times (the “discrepancies”). Since we not only have a ranking of insertion choices, but also a valuation on each branch (the value of Δ for each possibility of insertion), we may precisely control the occurrences of the discrepancies. Pure LDS(K) ([Harvey and Ginsberg, 1995]) would allow for discrepancies at any choice point, granted that all solutions are generated with less than K discrepancies. For dTP, this can be refined in several manners:

- one of the interests of Discrepancy-based Search is to generate solutions that are spread across the overall solution space. It is therefore particularly interesting to branch on options that are radically different at each discrepancy. Insertions at different positions within the same route are considered similar; therefore the alternate choices for insertion that are considered involve different routes;
- it is not worthwhile considering the second best possibility of insertion when its impact on the cost is significantly worse than the preferred choice. Branching on discrepancies should be reserved for situations of near-ties. Thus, a discrepancy is generated only when $\Delta_{k_2} \leq \beta \Delta_{k_1}$ where Δ_{k_1} is the best (least) insertion cost and Δ_{k_2} the second best, for some ratio $\beta \geq 1.0$;
- the branching scheme compares insertions within routes to which some clients have already been assigned with an insertion into one empty route (route creation). Comparing such options by their impact on the cost lower bound is sometimes unfair to the second option. Indeed, let i be the current client, $\Delta_{k_0} = c_{0,i} + c_{i,0}$ whereas $\Delta_{k_1} = c_{j_1,i} + c_{i,j_2} - c_{j_1,j_2}$ if the best place of insertion for i in k_1 is between clients j_1 and j_2 . Therefore, Δ_{k_0} accounts for two transitions between locations (back and forth between the depot, 0, and i), while Δ_{k_1} accounts for only one (replacing the direct transition from j_1 to j_2 by a transition from j_1 to i and one from i to j_2). In order to avoid such a bias against route creations, discrepancies where the alternate branch is a route creation are encouraged through a higher ratio threshold $\beta' > \beta$.

The overall Discrepancy-based Search *DISCREPANCY*(K), defined below, is a refined version of *LDS*(K). Its integer parameter, K , represents the maximal number of discrepancies allowed in a solution:

```

algorithm DISCREPANCY( $K$ ):
  sort all clients  $i$  by decreasing values of  $\text{priority}(i)$ 
  DISC(1,  $K$ )

```

```

procedure: DISC( $i, K$ ):
  if  $i > N$  return(success)
  else
    for  $k \in \text{domain}(\text{visitedBy}_i) \mid \exists j \neq i, \text{domain}(\text{visitedBy}_j) = \{k\}$ 
      try BEST_INS( $i, k$ )
    let  $k_1$  and  $k_2$  be the two branches with smallest  $\Delta_k$ 
    if  $\exists k_0 \in \text{domain}(\text{visitedBy}_i) \mid \forall j \neq i, \text{domain}(\text{visitedBy}_j) \neq \{k_0\}$ 
      try BEST_INS( $i, k_0$ )
    if no feasible solution was found
      return(failure)
    else
      branch on
        preferred choice:
          if  $\Delta_{k_1} < \Delta_{k_0}$  commit to BEST_INS( $i, k_1$ )
            DISC( $i + 1, K$ )
          else commit to BEST_INS( $i, k_0$ )
            DISC( $i + 1, K$ )
        if  $K > 0$  alternate branch:
          if  $\Delta_{k_0} \leq \Delta_{k_1}$ 
            if  $\Delta_{k_1} \leq \beta \Delta_{k_0}$  commit to BEST_INS( $i, k_1$ )
              DISC( $i + 1, K - 1$ )
            else if  $\Delta_{k_0} \leq \beta' \Delta_{k_1}$  commit to BEST_INS( $i, k_0$ )
              DISC( $i + 1, K - 1$ )
            else if  $\Delta_{k_2} \leq \beta \Delta_{k_1}$  commit to BEST_INS( $i, k_2$ )
              DISC( $i + 1, K - 1$ )

```

3.5 LS as Post-Optimization

This section illustrates various techniques for the exploration of constrained neighborhoods in the dTP case study. We assume to start with a feasible solution which can, for instance, be built through constructive methods presented in the previous section. We review four kinds of LS procedures on dTP: two simple adaptations of standard LS with constraint checks or inlined constraint checks and two descriptions of neighborhoods with CP, specifically, frozen fragments and CP models of the neighborhood.

Although we refer to the solution with the notations introduced in Section 3, it is never assumed that the variables of the model are CP domain variables. More precisely, in the two next sections (3.5.1 and 3.5.2), the easiest way to implement the LS methods consists in representing a solution with simple data structures (integer to represent variables modeling integer), whereas the approach in sections 3.5.3 and 3.5.4 require that the variables of the model are implemented as CP variables with domains and active propagation.

3.5.1 LS + constraint checks

A first idea for improving a solution to dTP consists in using classical VRP neighborhoods, and checking, for each neighbor, the feasibility with respect to side constraints. This method is illustrated on two simple neighborhoods, specifically *3-opt* and *node-transfer*.

3-opt. 3-opt is the straightforward extension of the 2-opt discussed in Section 2.1.1 in which three arcs, instead of two, are removed from the solution, and three others are added to obtain a new solution. Since this exchange is performed within the

same route, the feasibility test for the new solution needs to be executed only on that route. The best move is the one with the largest decrease of z (if any). An example of 3-opt move is depicted in Figure 1.

More formally, let k be a route, and i_1, i_2, i_3 be three different clients such that $visitedBy_{i_1} = visitedBy_{i_2} = visitedBy_{i_3} = k$, and $i_2 \in succ_{i_1}$ and $i_3 \in succ_{i_2}$. A new solution can be obtained by re-assigning variables $next_{i_1}, next_{i_2}$, and $next_{i_3}$, and keeping unchanged all the other $next$ variables. Since dTP incorporates side constraints, testing feasibility can be computationally expensive, but using constraints directly supports extensions to additional side constraints. For such 3-opt, since there is no exchange of clients among the routes, bin type and capacity constraints still hold, and only the time window constraints need to be checked.

The entire exploration of the neighborhood can be implemented as follows:

procedure: 3-opt_EXPLORE

for each $k \in \{1, \dots, M\}$

for each $i_1 \mid visitedBy_{i_1} = k$

for each $i_2 \mid visitedBy_{i_2} = k \wedge i_2 \in succ_{i_1}$

for each $i_3 \mid visitedBy_{i_3} = k \wedge i_3 \in succ_{i_2}$

let $j_1 := next_{i_1}, j_2 := next_{i_2}, j_3 := next_{i_3}$

try $next_{i_1} = j_2, next_{i_2} = j_3, next_{i_3} = j_1$

check feasibility for time windows

old $:= c_{i_1, j_1} + c_{i_2, j_2} + c_{i_3, j_3}$

new $:= c_{i_1, j_2} + c_{i_2, j_3} + c_{i_3, j_1}$

over all neighbors, minimize $\Delta := old - new$

Once the feasible solution with smallest Δ is found, the data structure $succ$ must be updated, so as the objective function z . As mentioned at the beginning of Section 3.5, we do not consider here CP variables, but the constraints (in this case the time window ones) are used off-line to check feasibility and update the $start$ variables.

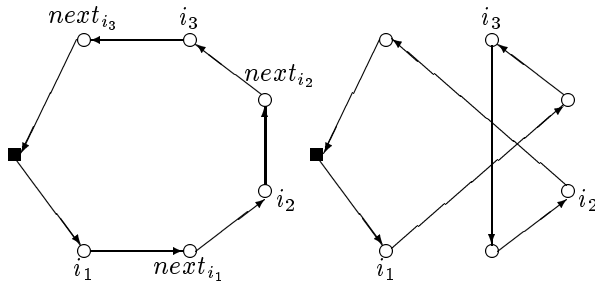


Figure 1: Example of 3-opt move.

Node-transfer. Node-transfer moves a single client from a route to another by keeping feasibility of both routes, and decreasing the overall value of z . This move leads to more complicated feasibility issues than 3-opt. Specifically, three constraints need to be checked for the route in which the client is inserted: bin type compatibility, bin capacity and the time window constraints. An example of node-transfer move is depicted in Figure 2.

More formally, let k_1 and k_2 be two routes, and let i_1, i_2 and i_3 be three clients such that $visitedBy_{i_1} = visitedBy_{i_2} = visitedBy_{i_3} = k_1$, and $next_{i_1} = i_2$, and $next_{i_2} = i_3$. Moreover, let j be one more client such that $visitedBy_j = k_2$ ⁶. A new solution is obtained by moving i_2 from k_1 to k_2 at position $next_j$.

⁶Since we do not have the variable $next_0$, if i_2 is the first client visited in route k_1 , and/or if we want to insert it as first client in route k_2 , then it is necessary to consider also variables $first_{k_1}$ and/or $first_{k_2}$.

A passive way of using constraints within a LS based on a single-client move is to have the following four nested loops:

```

procedure: node-transfer_EXPLORE
  for each  $k_1 \in \{1, \dots, M\}$ 
    for each  $i_2 \mid \text{visitedBy}_{i_2} = k_1$ 
      for each  $k_2 \in \{1, \dots, M\} \mid k_2 \neq k_1$ 
        for each  $j \mid \text{visitedBy}_j = k_2$ 
          try  $\text{visitedBy}_{i_2} = k_2, \text{next}_{i_2} = \text{next}_j,$ 
             $\text{next}_j = i_2, \text{next}_{i_1} = i_3$ 
          check feasibility for all constraints
          old :=  $c_{i_1, i_2} + c_{i_2, i_3} + c_{j, \text{next}_j}$ 
          new :=  $c_{i_1, i_3} + c_{j, i_2} + c_{i_2, \text{next}_j}$ 
        over all neighbors, minimize  $\Delta := \text{old} - \text{new}$ 

```

Unlike the 3-opt case, more side constraints need to be checked, since the move changes the client-truck assignments, and again an updating of the data structures is needed.

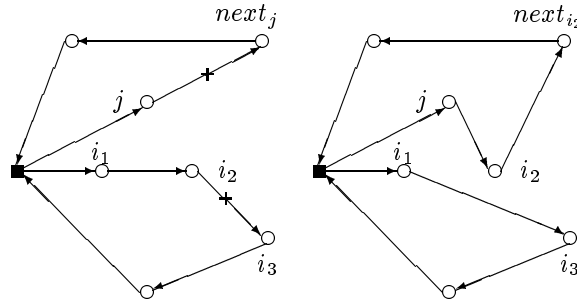


Figure 2: Example of node-transfer move.

3.5.2 Constraint checks within the neighborhood iteration

In the pseudo-code presented for the *node-transfer* neighborhood exploration much of the time is wasted generating trivially infeasible neighbors. For instance, it is useless iterating j over a route k_2 covered by a truck whose bins are not compatible with type_{i_2} . Therefore, to speed up the exploration, we can anticipate some of the feasibility checks within the nested loop and before the final constraint checks. The infeasibility (or ineffectiveness) of many of the solutions in the neighborhood can be easily detected by testing: (i) if the type of goods of client i_2 is compatible with the types collected by the bins of route k_2 , (ii) if the residual capacity of the bins of route k_2 can accommodate the quantity q_{i_2} , and (iii) if the neighbor improves z , i.e., if $c_{i_1, i_3} + c_{j, i_2} + c_{i_2, \text{next}_j} - c_{i_1, i_2} - c_{i_2, i_3} - c_{j, \text{next}_j} < 0$.

Such an optimized neighborhood exploration can be stated as follows:

```

procedure: node-transfer_EXPLORE_updated
  for each  $h_1 \in \{1, \dots, 2M\}$ 
    for each  $i_2 \mid \text{collectedIn}_{i_2} = h_1$ 
      for each  $h_2 \in \{1, \dots, 2M\} \mid$ 
         $h_2 \neq h_1 \wedge \text{collects}_{h_1} = \text{collects}_{h_2} \wedge \text{RC}_{h_2} \geq q_{i_2}$ 
        for each  $j \mid \text{collectedIn}_j = h_2$ 
          old :=  $c_{i_1, i_3} + c_{j, i_2} + c_{i_2, \text{next}_j}$ 
          new :=  $c_{i_1, i_2} + c_{i_2, i_3} + c_{j, \text{next}_j}$ 
          if  $\text{old} - \text{new} < 0$ 
            try  $\text{collectedIn}_{i_2} = h_2,$ 

```

$$\begin{aligned} next_{i_2} &= next_j, next_j = i_2, next_{i_1} = i_3, \\ visitedBy_{i_2} &= visitedBy_j \end{aligned}$$

check feasibility for time windows

over all such neighbors, minimize $\Delta := \text{old} - \text{new}$

where the RC_{h_2} represents the *residual capacity* of bin h_2 , i.e.,

$$RC_{h_2} = C - \sum_{i \mid collectedIn_i=h_2} q_i$$

and must be updated (together with *succ*, etc.) once a feasible (improving) move is performed. Not only does this exploration generate less infeasible neighbors, it also needs to check less constraints (only time window ones) on the outcome of the loop.

3.5.3 Freezing Fragments

This section describes another kind of moves called *shuffle* ([Applegate and Cook, 1991]) or *Large Neighborhood Search* (LNS, [Shaw, 1998]). Such moves keep a fragment of the current solution (freezing the values of variables in that fragment), forget the value and restore the initial domain of variables outside the fragment. Finding the best completion of such a state into a solution is an optimization subproblem. The search space for this subproblem is smaller than for the original problem because some variables have been fixed. Enforcing part of the assignments from the previous solution is the opposite process as relaxation: the problem is strengthened with additional constraints. The frozen fragments must be selected carefully and should be:

- large enough so that the subproblem is more tractable than the original one;
- but not too large so that the subproblem contains enough flexibility to improve over the current solution;
- good enough so that the subproblem consists in good solutions in the overall search space.

In a way, the fragment plays the role of a strictly followed heuristic while building a solution. However, since the fragment that is kept corresponds to choices that were made at any point in the search tree, and not necessarily at the root of the tree, such a destruction-construction process differs from chronological backtracking.

Such LNS algorithms can be easily implemented within CP systems, since the neighborhood exploration consists in solving the original problem while some variables are instantiated and the domain of others are reduced. Moreover, insertion algorithms are natural candidates to LNS. Specifically, clients are partitioned into two sets: a set of clients for which the route assignment and relative sequencing is kept (the fragment from the reference solution that is frozen), and the remainder of the clients, for which all decisions (assignment and sequencing variables) are forgotten.

As described in [Shaw, 1998], the forgotten part of the solution should be of relative significant size and should be made of related clients (clients from the same routes or from a given geographical area).

In the case of dPT, several neighborhoods can be proposed for such LNS:

- keeping all decisions concerning a subset of the *routes* and re-optimizing all the remainder of the solution;
- keeping all decisions concerning a subset of the *types of goods* and re-optimizing all the remainder of the solution;

- keeping all decisions concerning a subset of the *clients* and re-optimizing all the remainder of the solution.

Note that such different neighborhoods may be combined through VNS (see, [Mladenović and Hansen, 1997]).

In the following, a simple example of LNS based algorithm is proposed. The algorithm iteratively freezes a part of the problem and tries to extend the partial solution. The iterative process stops when a time limit is reached.

A solution s is encoded through three arrays $sNext$, $sSucc$, and $sVisitedBy$ containing the values of $next$, $succ$, and $visitedBy$ in s . The algorithm selects randomly a client i^* , and forgets the decisions that were made on the set of clients that are *close to* i^* . The term *close to* is quantitatively defined by the parameter $radius$, and the travel time matrix tt : two clients i, j are considered close if $tt_{ij} \leq radius$. The subproblem obtained is solved by the **GREEDY CONSTRUCTION** algorithm described in Section 3.4.2. Whenever an improving solution is found, the arrays encoding the solution must be updated. The overall algorithm is as follows:

algorithm: LNS($radius$)

```

while time is still available
  select at random a client  $i^*$ 
  shuffleSet :=  $\{j \mid tt_{i^*,j} \leq radius\}$ 
  restore initial domains for variables not in shuffleSet
  for all clients  $i \mid i \notin \text{shuffleSet}$ 
    visitedBy $_i$  = sVisitedBy $_i$ 
    if sNext $_i \notin \text{shuffleSet}$ 
      next $_i$  = sNext $_i$ 
    for all clients  $j \mid j \in sSucc_i$ 
      if  $j \notin \text{shuffleSet}$ 
         $j \in succ_i$ 
   $z < z^*$ 
  call algorithm GREEDY CONSTRUCTION
  if an improving solution is found
    for all clients  $i$ 
      sNext $_i$  := next $_i$ 
      sSucc $_i$  := succ $_i$ 
      sVisitedBy := visitedBy $_i$ 
     $z^* := z$ 
if any improving solution was found
  return(success)
otherwise return(failure)

```

Some interesting modifications of the LNS algorithm can be done.

- The $radius$ parameter can start with small values and increase whenever the greedy algorithm was unable to find improving solutions for a given number of consecutive times. Leading to a Variable Neighborhood Search, this modification will help escaping from local optimal solutions;
- Any constructive algorithm could be used to extend the frozen fragment. The use of Discrepancy-based Search algorithms could indeed increase the likelihood of finding solutions when the problem becomes harder due to the bounding constraint $z < z^*$;
- The pseudo code presented above forgot the assignments for all clients reachable within a given time from client i^* . This forgotten part of the solution could take into account a more sophisticated distance than mere travel time.

For instance, tt_{ij} could be multiplied by a weight depending on the routes, the types of goods, and the time windows of clients i and j . Clients having equal type of goods or overlapping time windows should be considered relatively *closer* than clients with different type of goods or distant time windows.

3.5.4 CP models for the Neighborhoods

In this section a neighborhood is modeled using CP as showed in Section 2.1.2; such a neighborhood is based on ejection chains: a set of clients belonging to different routes are chosen and re-located (see Figure 3).

A given solution s is encoded through four arrays $sNext$, $sSucc$, $sVisitedBy$, and $sPrev$; the first three arrays contain the values of $next$, $succ$, and $visitedBy$ in solution s ; array $sPrev$ contain the inverse of $next$ (if $next_i = j$ in s , then $sNext_i = j$, $sPrev_j = i$).

The following neighborhood structure can be defined: a move from s chooses $K \leq M$ clients i_1, i_2, \dots, i_K such that all the visits are performed in different trucks in solution s . Then it removes the edges $(sPrev_{i_j}, i_j)$, $(i_j, sNext_{i_j})$, with $j = 1, \dots, K$, and re-connects the tours by rotating the nodes i_1, i_2, \dots, i_K . For all $j = 1, \dots, K - 1$, i_{j+1} replaces i_j ; the rotation is closed by replacing i_K with i_1 . We may either look for any improving move or for the best move in the neighborhood.

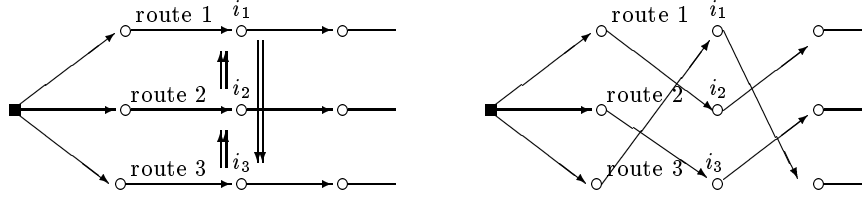


Figure 3: Example of ejection chain.

This neighborhood structure can be defined using CP by adding the following model to the existing dTP model:

$$\begin{aligned}
& \min && dtCost \\
& \text{on} && \\
& && dtCost < 0 \\
& \forall j \in \{1, \dots, M\} && I_j \in [-1, 1..N] \\
& && size \in [2..M] \\
& \text{subject to} && \\
& \forall i, j \in \{1, \dots, M\}, i < j && i \leq size \Leftrightarrow \\
& && \quad sVisitedBy[I_i] \neq sVisitedBy[I_j] \quad (15) \\
& \forall j \in \{1, \dots, M\} && j > size \Leftrightarrow I_j = -1 \quad (16) \\
& \forall j \in \{1, \dots, M\}, i \in \{1, \dots, N\} && I_j = i \Rightarrow \\
& && \quad next[sPrev_i] = \ell, \forall \ell \mid i \in sSucc_\ell \quad (17) \\
& \forall j \in \{1, \dots, M\}, i \in \{1, \dots, N\} && I_j = i \Rightarrow \\
& && \quad next[\ell] = sNext_\ell, \forall \ell \mid \ell \in sSucc_i \quad (18) \\
& \forall j \in \{1, \dots, M - 1\} && j < size \Rightarrow next[sPrev[I_j]] = I_{j+1} \quad (19) \\
& \forall j \in \{1, \dots, M - 1\} && j < size \Rightarrow next[I_{j+1}] = sNext[I_j] \quad (20) \\
& && next[sPrev[I_{size}]] = I_1 \quad (21) \\
& && next[I_1] = sNext[I_{size}] \quad (22)
\end{aligned}$$

$$\begin{aligned}
dtCost = & \\
& \sum_{i=1}^{size-1} (c[sPrev[I_i], I_{i+1}] + c[I_{i+1}, sNext[I_i]]) + \\
& c[sPrev[I_{size}], I_1] + c[I_1, sNext[I_{size}]] - \\
& \left(\sum_{i=1}^{size} (c[sPrev[I_i], I_i] + c[I_i, sNext[I_i]]) \right) \tag{23}
\end{aligned}$$

A solution of the neighborhood model defines an ejection chain affecting $size$ clients; since $size$ can take any value between 2 and M , we introduce M variables representing the variable length ejection chain. The first $size$ variables identify the clients changing route while the others are meaningless and are set to -1 .

M domain variables I_1, I_2, \dots, I_M , are used to identify the clients. The domain of variable I_j is $[-1, 1..N]$; the first $size$ variables⁷ will take values different from -1 . Client I_2 goes in place of client I_1 , client I_3 goes in place of client I_2 , etc. finally client I_1 goes in place of client I_{size} . All variables I_j with $j > size$ take the value -1 . The objective function of the neighborhood model is the difference between the cost of the current solution s and the cost of the tentative solution.

Constraints (15) and (16) define the neighborhood structure: $size$ consecutive I_j variables identify the clients in the ejection chain, and the remaining $M - size$ are constrained to be equal to -1 . The clients must be chosen from $size$ different trucks (15).

Constraints (17)-(22) define the interface constraints between the dTP model and the neighborhood model. They are basically of two types: some of them are devoted to restoring of the unchanged part of the solution (17)-(18), some others are devoted to encoding the move (19)-(22). Finally, constraint (23) links the neighborhood variables to $dtCost$.

Defining the neighborhood structure via a CP model yields a stable neighborhood model that would remain valid if side constraints (such as precedence constraints, etc.) were added to dTP. All side constraints propagate from the problem variables to the neighborhood variables, removing some neighbors. For example, once variable I_1 is instantiated to a value i_1 visited by the first truck, all clients i_2 requiring a capacity exceeding the remaining capacity of the first truck are automatically removed from the domain of variable I_2 by propagation of the interface constraints.

3.6 LS during construction

The last two sections showed how constructive algorithms from greedy to Discrepancy-based Search could yield one or several solutions, and how local moves from various neighborhoods could improve them. This section concerns the use of local moves at the very heart of the construction process. Following [Caseau and Laburthe, 1999], we develop on the idea of Incremental Local Optimization (ILO, see Section 2.2.7) which applies improving local moves after each construction step.

3.6.1 Single Route Optimization

A standard move in routing problems consists in optimizing the sequence of visits for each route. Optimizing a route yields one small constrained TSP per route. This leads to a decomposition of the problem: the partition of clients by routes is kept, and the induced independent TSPs are re-solved. Each neighborhood for

⁷When I_j is used to index an array *array*, and -1 is part of the domain of I_j , the value $(array)[-1]$ is conventionally considered equal to -1 .

the TSP induces a neighborhood for dTP. A solution to dTP may, for instance, be transformed into a neighbor one by applying on any of the routes:

- a feasible 3-opt move ([Reinelt, 1994]);
- a feasible Lin and Kernighan move ([Lin and Kernighan, 1973]);
- a sequence of improving feasible 3-opt moves until no more such moves can be found;
- a complete algorithm that sequences each route to optimality.

The last case is of particular interest. Indeed, many routing problems, although involving thousands of clients typically assign less than 15 clients per truck. It is thus easy to solve to optimality the constrained TSP associated to a route by branch and bound with constraint propagation. Such an approach was shown to be viable in [Caseau and Laburthe, 1996] and [Focacci et al., 1999b].

After each insertion, the partial solution can be re-optimized in a state where each route is optimally sequenced. For the decomposition of dTP into a master partitioning sub-problem and an induced sequencing (routing) sub-problem, the sequencing sub-problem is solved to optimality. The complexity of the overall procedure remains tractable because the LS phase that is performed at each insertion step can be done much faster than a LS phase that would be performed in the end, after a complete solution has been built. Indeed, after the insertion of a node i in route k ($visitedBy_i = k$), TSP moves should be applied on route k only. Therefore, it is worthwhile resolving the TSP for route k only. This fast incremental exploration accounts for the overall efficiency of applying local optimization within the construction process.

3.6.2 Ejection Chains

The neighborhood introduced above is well suited for improving sequencing decisions among clients on the same route but it never questions the partition of clients into routes. Since the insertion algorithm inserts clients one at a time, it is likely to make assignments (of clients to routes) that seem good when only a subset of the clients is considered, but that later prove to be suboptimal when further clients are inserted. This section presents a neighborhood whose aim is to repair such partitioning decisions that turn out to be wrong along the insertion process.

Section 3.5.4 proposed a CP model for an ejection chain neighborhood. The idea of ejection chains, a standard technique for packing problems, is the following. Let i_1, \dots, i_r be a set of r clients currently assigned to different routes (say, $visitedBy_{i_1} = k_1, \dots, visitedBy_{i_r} = k_r$) and i_0 be another client not assigned to k_1 . The move changes the bin assignment into $visitedBy_{i_0} = k_1, visitedBy_{i_1} = k_2, \dots, visitedBy_{i_{r-1}} = k_r$ and resolves the induced sequencing problems on routes k_1 to k_r . Applied on the dTP, this move amounts to change the route assignment of a few nodes while respecting all constraints (satisfying capacity constraints with the new insertion $visitedBy_{i_j} = k_{j+1}$ is eased by the removal of i_{j+1} from k_{j+1}).

Such moves are particularly interesting in the case of promising infeasible insertions. For instance, one may foresee that the insertion $visitedBy_i = k$ would yield a very small insertion cost Δ_k , but that it is infeasible because of, say, a capacity constraint (there is no more room for client i in the route k). Instead of considering the insertion of i into routes that are geographically farther away (which may cause significant detours, thus higher insertion cost Δ), it is interesting to try to insert i into the route k by removing another client j from route k and transferring it to another route. This leads to the search for ejection chains initiated by the insertion of i in k . Such moves are particularly useful when capacity constraints on the

routes are tight. Another situation where ejection chains prove useful is the case of travel optimization under a small number of trucks. In such a case, the routes must be tight in order to cover all clients. What usually happens in the insertion process is that the algorithm is not able to insert some of the clients towards the end of the construction (all tentative insertions turn out to be infeasible). Ejection chains can be used as a way to force these insertions: [Caseau et al., 1999]. An alternate chain of insertions and ejections is searched for such that i can be inserted into some route k_1 by transferring another client j from k_1 to k_2 , and so on. Such neighborhoods can be explored through Breadth First Search in order to find the least length ejection chain (the one that involves the least number of clients), likely to be less disruptive of the current solution.

Finally, one should be careful about the fact that such ejection chain neighborhoods are very large neighborhoods and that their exploration may take significantly longer than the other neighborhoods presented so far (3-opt, single route direct optimization, etc.). It amounts to a form of LNS and should be used with care. Nevertheless, it complements very efficiently the short-sightedness nature of constructive algorithms which build the solution one step at a time by repairing some of the early non-optimal route assignments.

4 Conclusions

This chapter has presented a variety of techniques for using local search methods with constraints. Hybrid combinations have been described for local search methods as well as for constructive global search methods.

Constraints can be blended with local search procedures in several ways:

- by expressing the problem as a standard problem with additional side constraints, iterating a neighborhood for the standard problem and, for each neighbor, checking feasibility for all side constraints; or by optimizing the neighborhood exploration procedure with inlined constraint checks. These methods can either be implemented with any programming language following the ideas described in this paper, or with a CP language using already defined specific data structures (such as constraints, neighborhood objects, and move operators) ([De Backer et al., 2000]).
- by using global search techniques for exploring the neighborhood defined by a fragment of the current solution; these methods are also described in [Shaw, 1998, Caseau et al., 1999] for routing problem, or in [Nuijten and Le Pape, 1998] for scheduling problems.
- by defining the search for the best neighbor as a constrained optimization problem on its own; originally introduced in [Pesant and Gendreau, 1996, Pesant and Gendreau, 1999], it has been applied mainly to timetabling and routing problems in [Pesant and Gendreau, 1996, Pesant and Gendreau, 1999, Pesant et al., 1997].

Local search techniques can be introduced within a constructive global search algorithm with the following techniques:

- Restricted Candidate Lists to filter the good branches of a choice point (see [Cesta et al., 2000] for a recent application to scheduling problems);
- Discrepancy-based Search to generate near-greedy paths in a search tree;
- Incremental Local Optimization to improve the partial solutions at each step of a construction process such as a greedy insertion algorithm.

In all these cases, CP provides the user with clean formalism to combine technologies: use propagation to reduce the size of neighborhoods, use global search techniques to explore a neighborhood, control the divergence of a construction process from a greedy path, etc. In a sense, CP supports a clean engineering of many “good old tricks” from Operations Research. The first clean integration has been the expression of neighborhoods with CP models. CP models could be introduced for many other algorithm engineering purposes such as objective function combinations for multi-criteria optimization (see, e.g., [Focacci et al., 2000a]), or solution fragment combinations for population-based optimization methods. All these are open research topics. The definite impact of CP to Operations Research in general, and to LS in particular, is the introduction of structuring objects that provide a way to easily combine techniques.

Many languages embedding constraint propagation have been developed by research institutes, universities, and private companies; knowing that we are risking appearing unfair with respect to some of them, we believe it is important to give some pointers to few tools that demonstrated larger acceptance both in industries and academics. A practitioner desiring to experiment local search and constraint programming could take a closer look to CP tools such as CHIP ([Aggoun and Beldiceanu, 1992]), CHOCO ([Laburthe, 2000]), Eclipse ([Schimpf et al., 1997]), and ILOG Solver ([Solver, 2000]). All of them provide a constraint propagation engine, together with tree search exploration methods, and facilities to design local search methods.

Finally, it is worthwhile mentioning that recently the CP community has more and more showed interest in hybrid approaches for solving optimization problems. Some of the conferences and workshops that regularly present interesting papers on the subject are the *International Conference on Principles and Practice of Constraint Programming* (CP), the *International Workshop on Integration of Artificial Intelligence and Operations Research Techniques in CP for Combinatorial Optimization Problems* (CP-AI-OR), and the Constraint clusters at *INFORMS* conferences.

Acknowledgments

We warmly thank Paul Shaw, Michela Milano, Fred Glover, Eric Bourreau, Etienne Gaudin, Cesar Rego, Gilles Pesant and Benoît Rottembourg for reading preliminary versions of this work and for many interesting discussions on the topic.

References

- [Aggoun and Beldiceanu, 1992] Aggoun, A. and Beldiceanu, N. (1992). Extending CHIP in order to solve complex scheduling and placement problems. In *Actes des Journees Francophones de Programmation et Logique*, Lille, France.
- [Applegate and Cook, 1991] Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156.
- [Beldiceanu et al., 1999] Beldiceanu, N., Bourreau, E., Simonis, H., and Rivrau, D. (1999). Introducing metaheuristics in CHIP. In *Proceedings of the 3rd Metaheuristics International Conference*, Angra do Reis, Brazil.
- [Caprara et al., 1997] Caprara, A., Fischetti, M., Toth, P., Vigo, D., and Guida, P.-L. (1997). Algorithms for railway crew management. *Mathematical Programming*, 79:125–141.

- [Caseau and Laburthe, 1996] Caseau, Y. and Laburthe, F. (1996). Improving branch and bound for job-shop scheduling with constraint propagation. In Deza, M., Euler, R., and Manoussakis, Y., editors, *Proceedings of Combinatorics and Computer Science, CCS'95, LNCS 1120*. Springer-Verlag, Berlin Heidelberg.
- [Caseau and Laburthe, 1999] Caseau, Y. and Laburthe, F. (1999). Heuristics for large constrained routing problems. *Journal of Heuristics*, 5:281–303.
- [Caseau et al., 1999] Caseau, Y., Laburthe, F., and Silverstein, G. (1999). A meta-heuristic factory for vehicle routing problems. In Jaffar, J., editor, *Principle and Practice of Constraint Programming - CP'99, LNCS 1713*, pages 144–158. Springer-Verlag, Berlin Heidelberg.
- [Cesta et al., 2000] Cesta, A., Oddi, A., and Smith, S. (2000). A constraint-based method for project scheduling with time windows. *Journal of Heuristics*. To appear.
- [De Backer et al., 2000] De Backer, B., Furnon, V., Shaw, P., Kilby, P., and Prosser, P. (2000). Solving vehicle routing problems using constraint programming and meta-heuristics. *Journal of Heuristics*, 6:481–500.
- [Dell'Amico and Trubian, 1993] Dell'Amico, M. and Trubian, M. (1993). Applying tabu-search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252.
- [Feo and Resende, 1995] Feo, T. and Resende, M. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.
- [Focacci et al., 2000a] Focacci, F., Laborie, P., and Nuijten, W. (2000a). Solving scheduling problems with setup times and alternative resources. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling, AIPS'00*. AAAI Press.
- [Focacci et al., 1999a] Focacci, F., Lodi, A., and Milano, M. (1999a). Cost-based domain filtering. In Jaffar, J., editor, *Principle and Practice of Constraint Programming - CP'99, LNCS 1713*, pages 189–203. Springer-Verlag, Berlin Heidelberg.
- [Focacci et al., 1999b] Focacci, F., Lodi, A., and Milano, M. (1999b). Solving TSP with time windows with constraints. In De Schreye, D., editor, *Logic Programming - Proceedings of the 1999 International Conference on Logic Programming*, pages 515–529. The MIT-press, Cambridge, Massachusetts.
- [Focacci et al., 2000b] Focacci, F., Lodi, A., Milano, M., and Vigo, D. (2000b). An introduction to constraint programming. *Ricerca Operativa*, 91:5–20.
- [Gendreau et al., 1992] Gendreau, M., Hertz, A., and Laporte, G. (1992). New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40:1086–1094.
- [Glover, 1995] Glover, F. (1995). Tabu thresholding: Improved search by nonmonotonic trajectories. *ORSA Journal on Computing*, 7:426–442.
- [Golden and Assad, 1988] Golden, B. and Assad, A. (1988). *Vehicle Routing: Methods and Studies*. North-Holland, Amsterdam.

- [Haralick and Elliott, 1980] Haralick, R. and Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.
- [Harvey, 1995] Harvey, W. (1995). *Nonsystematic Backtracking Search*. PhD thesis, Stanford University.
- [Harvey and Ginsberg, 1995] Harvey, W. and Ginsberg, M. (1995). Limited discrepancy search. In *Proceedings of the 14th IJCAI*, pages 607–615. Morgan Kaufmann.
- [Junker, 2000] Junker, U. (2000). Preference-based search for scheduling. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence - AAAI-2000*, pages 904–909.
- [Kindervater and Savelsbergh, 1997] Kindervater, G. and Savelsbergh, M. (1997). Vehicle routing: Handling edges exchanges. In Aarts, E. and Lenstra, J. K., editors, *Local Search in Combinatorial Optimization*, pages 337–360. J. Wiley & Sons, Chichester.
- [Laburthe, 2000] Laburthe, F. (2000). CHOCO: implementing a CP kernel. In *CP'00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems - TRICS*. Singapur.
- [Lin and Kernighan, 1973] Lin, S. and Kernighan, B. (1973). An effective heuristic for the traveling salesman problem. *Operations Research*, 21:498–516.
- [Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.
- [Marriott and Stuckey, 1998] Marriott, K. and Stuckey, P. (1998). *Programming with Constraints*. The MIT Press.
- [Mautor and Michelon, 1997] Mautor, T. and Michelon, P. (1997). MIMAUSA: A new hybrid method combining exact solution and local search. In *Proceedings of the 2nd International Conference on Meta-Heuristics*, Sophia-Antipolis, France.
- [Michel and van Hentenryck, 1997] Michel, L. and van Hentenryck, P. (1997). Localizer: A modeling language for local search. In Smolka, G., editor, *Principle and Practice of Constraint Programming - CP'97, LNCS 1330*, pages 237–251, Berlin Heidelberg. Springer-Verlag.
- [Minton et al., 1992] Minton, S., Johnston, M., Philips, A., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205.
- [Mladenović and Hansen, 1997] Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24:1097–1100.
- [Nuijten, 1994] Nuijten, W. (1994). *Time and Resource Constrained Scheduling, a Constraint Satisfaction Approach*. PhD thesis, University of Eindhoven, The Netherlands.
- [Nuijten and Le Pape, 1998] Nuijten, W. and Le Pape, C. (1998). Constraint based job shop scheduling with ILOG scheduler. *Journal of Heuristics*, 3:271–286.

- [Pesant and Gendreau, 1996] Pesant, G. and Gendreau, M. (1996). A view of local search in constraint programming. In Freuder, E., editor, *Principle and Practice of Constraint Programming - CP'96, LNCS 1118*, pages 353–366. Springer-Verlag, Berlin Heidelberg.
- [Pesant and Gendreau, 1999] Pesant, G. and Gendreau, M. (1999). A constraint programming framework for local search methods. *Journal of Heuristics*, 5:255–279.
- [Pesant et al., 1998] Pesant, G., Gendreau, M., Potvin, J., and Rousseau, J. (1998). An exact constraint logic programming algorithm for the travelling salesman problem with time windows. *Transportation Science*, 32:12–29.
- [Pesant et al., 1997] Pesant, G., Gendreau, M., and Rousseau, J.-M. (1997). GENIUS-CP: A generic single-vehicle routing algorithm. In Smolka, G., editor, *Principle and Practice of Constraint Programming - CP'97, LNCS 1330*, pages 420–433. Springer-Verlag, Berlin Heidelberg.
- [Prais and Ribeiro, 1998] Prais, M. and Ribeiro, C. (1998). Reactive grasp: An application to a matrix decomposition problem in TDMA traffic assignment. Technical report, Catholic University of Rio de Janeiro, Department of Computer Science.
- [Prestwich, 2000] Prestwich, S. (2000). A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In Dechter, R., editor, *Principle and Practice of Constraint Programming - CP2000, LNCS 1894*, pages 337–352. Springer-Verlag, Berlin Heidelberg.
- [Régin, 1994] Régin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence - AAAI'94*, pages 362–367.
- [Reinelt, 1994] Reinelt, G. (1994). *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag.
- [Russell, 1995] Russell, R. (1995). Hybrid heuristics for the vehicle routing problem with time windows. *Transportation Science*, 29:156–166.
- [Schimpf et al., 1997] Schimpf, J., Novello, S., and Sakkout, H. (1997). *IC-Parc ECLiPSe Library Manual*.
- [Selman and Kautz, 1993] Selman, B. and Kautz, H. (1993). Domain-independent extension to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI-93, 13th International Joint Conference on Artificial Intelligence*, pages 290–295, Sidney, AU.
- [Selman et al., 1992] Selman, B., Levesque, H., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California. AAAI Press.
- [Shaw, 1998] Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M. and Puget, J.-F., editors, *Principle and Practice of Constraint Programming - CP'98, LNCS 1520*, pages 417–431. Springer-Verlag, Berlin Heidelberg.
- [Shaw et al., 2000] Shaw, P., Furnon, V., and De Backer, B. (2000). A lightweight addition to CP frameworks for improved local search. In *Proceedings of CP-AI-OR'00*, Paderborn, Germany.

- [Solver, 2000] Solver (2000). *ILOG Solver 5.0 User's Manual and Reference Manual*. ILOG, S.A.
- [Toth and Vigo, 2002] Toth, P. and Vigo, D. (2002). *The Vehicle Routing Problem*. Monographs on Discrete Mathematics and Applications. SIAM.
- [van Hentenryck et al., 1993] van Hentenryck, P., Saraswat, V., and Deville, Y. (1993). and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University.
- [Walser, 1999] Walser, J. (1999). *Integer Optimization by Local Search*, volume 1637 of *Lecture Notes in Artificial Intelligence*. Springer Verlag.
- [Walsh, 1997] Walsh, T. (1997). Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence - IJCAI*. Morgan Kaufmann.