Constraint-Based Very Large-Scale Neighborhood Search

Sébastien Mouthuy

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

September 2011

Information and Communication Technologies, Electronics and Applied Mathematics Institute
Louvain School of Engineering
Louvain-la-Neuve
Belgium

Thesis Committee:

Yves Deville (co-director)	Université Catholique de Louvain
Pascal Van Hentenryck (co-director)	Brown University
François Glineur	Université Catholique de Louvain
Thomas Stützle	Université Libre de Bruxelles
Manuel Iori	Università degli Studi di Modena e Reggio Emilia
Charles Pecheur	Université Catholique de Louvain
Olivier Bonaventure (president)	Université Catholique de Louvain

Abstract

A combinatorial optimization problem requires to take discrete decisions under constraints and optimizing a given objective function, such as planning, routing and scheduling problems. These problems arise in many industrial fields and are of economical significance.

The general scope of this thesis is about Local Search (LS), that is an iterative methodology to solve combinatorial optimization problems. The principle is to generate a first solution, and to iteratively perform slight modifications to this solution in order to obtain a good score with respect to the objective function.

Very Large-Scale Neighborhood (VLSN) is a sophisticated technique in Local Search to perform many modifications to the solutions at each iteration. These techniques have a greater visibility at each iteration and choose the next solution more efficiently. Very Large-Scale Neighborhoods have been successfully applied on many complex real-life problems. However VLSN are very hard to implement. This prevents the applicability of these techniques to new problems.

This thesis aims at remedying this limitation and presents a framework that expresses VLSN search algorithms in terms of high-level components. VLSN search algorithms expressed by mean of our framework exhibits several benefits compared to existing approaches: 1) a more natural design of VLSN search algorithm, 2) a better reusability of existing components, 3) a greater adaptability to modifications to the problem to solve, and 4) a faster development of complex VLSN approaches.

As a proof of concept, we implemented this framework. Experimental results show that our approach is comparable in time with respect to existing approaches, and that it allows to obtain state-of-the-art solutions on two real-life problems exhibiting different structure (one timetabling and one routing application).

This validates that our approach is a helpful and efficient support to develop VLSN search algorithms on new and complex problems.

Acknowledgments

I want to warmly thank Yves Deville who has been my advisor for both my master and PhD thesis. His accurate questions and his constant will to make things as clear as possible often transformed an intuitive idea into a strong concept. You always have the right feeling to gather people and to make everyone build strong relationships. This is unvaluable for all of us. Thank you Yves!

My thanks also goes to Pascal Van Hentenryck, my co-advisor. I had wonderful discussions with him when he invited me in his lab. Your vision greatly guided me when it was not clear where to dig. Thank you Pascal!

I also want to thank all the members of the BeCool group who have been so nice to spend time with. Our many discussions have always been funny, meaningful and respectful of our differences. I wish to all of you very good times and great success in life. I also want to thank everyone in the INGI department for their readiness to share a good joke.

I am grateful to the members of my accompaniment commitee, Thomas Stützle and François Glineur for their helpful feedback on my work, and to the whole jury for their meaningful reading of this thesis.

Special thanks go to my family, my friends and all the people who have been so nice with me. We are all made to work hard to realize projects we believe on, but these are not self-sustainable. Warm love supports all we are and all we do.

This research has been suppported by the Fond National de la Recherche Scientifique (FNRS).

Contents

Part I

Background

Chapter 1

Introduction

This thesis is concerned with solving Combinatorial Optimization Problems using Very Large-Scale Neighborhood (VLSN) search, a class of local-search algorithms whose neighborhoods contain an exponential number of neighbors.

A combinatorial optimization problem (COP) requires to assign values from a finite set to variables, such that a set of constraints are respected and an objective function is minimized. Many real-life and industrial problems are combinatorial optimization problems such as planning, routing and scheduling problems. Main combinatorial optimization problems are NP-hard; it is strongly believed that there is no polynomial time algorithm solving them to optimality or, in some cases, even approximatively. There exist several techniques to find solutions to this type of problems that perform well in practice. Local Search is one of these techniques.

Local Search (LS) is an iterative methodology to solve combinatorial optimization problems. A local search algorithm only considers assignments of the variables that respect the constraints of the problem. These assignments are called solutions. A Local Search algorithm begins with a first solution, and then iteratively performs slight modifications, called moves, to the current solution. By applying such moves, Local Search algorithms hopefully reach a solution having a good score with respect to the objective function. The set of solutions that can be reached by performing a move on a given solution is called the neighborhood of this solution. In traditional Local Search, the neighborhoods are polynomially sized. Then a search algorithm selects which candidate in the neighborhood has to be chosen at each iteration. Generally, in local

search approaches, each move is explicitely considered in order to find the move leading to a solution that minimizes the objective function among the neighbors.

One important aspect in the design of Local Search algorithms is how it deals with local optima. A local optimum is a solution such that no move leads to an improved solution wrt the objective function. The set of local optima thus depends on the neighborhood used. There exist several solutions to this problem, but no one universally works, and dealing with local optima is always problematic. A rule of thumb of local search is that the bigger the neighborhood used in a Local Search approach, the better it performs as there are less local optima. Very Large-Scale Neighborhoods were designed with this rule in mind.

Constraint-Based Local Search is a technology that performs local search on high-level models, enabling the implementation of very efficient and complex LS algorithms for solving complex Combinatorial Optimization Problems. The main added-value of this approach is the ability to quickly validate many different ideas. This is crucial to efficiently solve a new problem. Indeed Combinatorial Optimization is a field such that great ideas have a huge impact on efficiency (far more than code optimization), but has very few tools to predict how an idea will speed-up the Local Search algorithm. By enabling fast prototyping, CBLS helps in finding the best algorithms for solving a problem.

The current stage of development of Constraint-Based Local Search only provide support for standard neighborhoods, i.e. polynomially sized neighborhoods. This is one of the main limitations of this technology.

Very Large-Scale Neighborhoods (VLSN) are neighborhoods that contain an exponential number of neighbors. By considering neighborhoods of exponential size, VLSN search often produces local optima of higher quality than polynomial-sized neighborhoods. VLSN usually have a structure such that the best neighbor wrt the objective function can be computed in polynomial time. They have been used to successfully solve many combinatorial optimization problems.

Until now, all VLSN approaches that are efficient in practice have been designed for a given problem, a particular VLSN and a specific search algorithm. Efficient VLSN approaches usually rely on Improvement Graphs and sophisticated ad-hoc algorithms that are closely related to (1) the problem, (2) VLSN and (3) search algorithm studied. Such results raise the following questions: what if we want to slightly change one of these three components? Could we still reuse the others?

How should we modify the other components? The implicit answer given by existing papers was that the problem, the VLSN and the search algorithm have all to be known perfectly in order to design a VLSN approach that performs well in practice.

Current VLSN theories do not lead to an efficient generic implementation of a VLSN search algorithm. The state of knowledge about VLSN is not mature enough to adapt existing VLSN techniques to new problems. The design of VLSN approaches still requires a lot of knowledge about the problem in order to achieve good experimental results. The development time and the amount of theoretical and technical knowledge are prohibitive to fit the flexibility required by real-life environments.

The objective of this research is to integrate Very Large-Scale Neighborhoods into Constraint-Based Local Search. These two technologies may then benefits from the strengths of each other to counterbalance their weaknesses. First CBLS may benefit from VLNS by supporting larger neighborhoods, enabling to avoid to get trapped in poor local optima. Second, VLSN search algorithms may be quickly prototyped by being expressed in terms of reusable components. VLSN search algorithms may also be reused on different problems by supporting the transparent addition of constraints or objective functions to the model. Finally the power of VLSN may be used with the flexible support of meta-heuristics in CBLS.

Our approach to fulfill this objective is threefold. The first step undertaken in this perspective was to identify (1) which problems may be solved using a VLSN search algorithm, (2) the exact properties of the structure of the existing VLSN that enables to search them very efficiently, and (3) how to implement an algorithm that searches a given VLSN efficiently.

The second step of this research expressed the problem to solve, the structure of the VLSN and the search algorithm as separated components by designing high-level abstractions. This step followed the objective of clearly identifying the exact properties that are important in the design of a VLSN search algorithm. This step was motivated by two recent works. The first one is Constraint-Based Local Search (CBLS) that enables to model the problem and, based on this model, to describe a local search algorithm. The major theoretical result of this work is the definition of differential invariants that capture the core properties of the problem that have to be known by a local search algorithm to solve it. The research trying to automatically derive the algorithm searching

the VLSN from its structure also motivated this step although they did not lead to efficient practical approaches.

The third step was to implement this theoretical architecture and to validate its efficiency in practice. This step was necessary to show that the high-level concepts designed previously were capturing the essential properties required to design an efficient VLSN search algorithm. Three main problems were considered. The Capacitated Minimum Spanning Tree allows to precisely assess the efficiency of our implementation compared to state-of-the-art VLSN search algorithms. This first problem also illustrates how the structure of VLSN may be modified without affecting the two other components (the problem and the search algorithm). The Capacitated Exam Timetabling Problem illustrates the inherent adaptability of our abstractions to changes to the problem. The Vehicle Routing Problem with Time Windows illustrates the benefits of our approach in terms of reuse, flexibility and efficiency when designing efficient VLSN search algorithms for solving complex real-life applications.

The contributions of this thesis are meaningful both for the academic and industrial communities. This thesis introduces several new concepts or redefines existing ones more abstractly in order to express VLSN and search algorithms generically.

- The concept of **independence** states when several moves can be applied together without interfering with each other
- The concept of **MoveGraph** enables to describe VLSN independently of the problem to solve
- **Compositionality** states which objective functions and constraints allow to solve a problem by means of VLSN search algorithms. It also clearly identifies the properties of the problem required to be solved efficiently by VLSN approaches
- **Improvement graphs** can be automatically derived, and incrementally updated, from a MoveGraph and a CBLS model of the problem to solve, that can be defined independently from each other
- **Input and output variables** are defined for a given objective function or constraint and enable the automatic computation of compositionality and independence. The input and output variables are the key components of our framework that allows to

separate the definition of the problem and the search algorithm; the search algorithm computing the best candidate in a VLSN can be defined independently of the problem to solve

- **Further developments of VLSN techniques** are enabled by this high-level design of VLSN approaches

These theoretical concepts enable to implement a framework that raises VLSN techniques to a high-level of expressive abstractions. In our approach, a VLSN local search algorithm is expressed as three separated components: the model (describing what is a solution to the problem), the VLSN (what are the neighbors of a solution) and the search algorithm (efficient algorithm computing the best neighbor). This existing framework has several contributions:

- VLSN techniques are now **modular**: each component can be modified without any need to change the two others. This makes experimentation easier and faster, which is critical to efficiently solve NP-hard problems.
- **Reusability** of VLSN implementations: existing components can be reused to quickly implement a VLSN search algorithm
- This work provides a **library** of such components to solve partitioning, permutation and vehicle routing problems that can be reused on a large set of problems
- The implementation of VLSN algorithms is now more **time-efficient**: the complex algorithmic aspects of VLSN approaches are hidden inside the components and achieving state-of-the-art results does not require much effort
- Our approach **broadens VLSN usage** to non-specialists as one can reason in terms of high-level concepts while not considering implementation issues

The structure of this thesis is detailled hereafter. Chapter 2 introduces the basic concepts of Local Search and Constraint-Based Local Search used throughout this work. Chapter 3 describes the Very Large-Scale Neighborhoods (VLSN) and gives some theoretical insights why VLSN are so efficient in practice. The state-of-the-art papers about VLSN are presented at the end of this chapter. Chapter 4 describes our VLSN theory by defining the high-level concepts abstracting the VLSN. Chapter 5 describes how a natural extension of the CBLS framework

implements our VLSN theory and assesses the relevance of our theory. Chapter 6 presents how the Capacitated Minimum Spanning Tree problem can be solved using the tools developed in the previous chapter and validates the efficiency of our implemented approach both in terms of time and quality of the solutions found. Chapter 7 solves the Capacitated Examination Timetabling Problem and illustrates the inherent adaptability of our abstractions to changes to the problem. Chapter 8 presents the benefits of our approach when developing efficient VLSN search algorithm on complex real-life applications by successfully solving the Vehicle Routing Problem with Hard/Soft Time-Windows. Finally Chapter 9 highlights the findings of this research, their added value for the academic and industrial communities and identifies the perspectives opened by this work.

Chapter 2

Preliminaries

2.1 Combinatorial Optimization Problem

A Combinatorial Optimization Problem (COP) is the problem of assigning values to a set of variables where constraints specify that some subsets of values are forbidden. In addition, an objective function quantifies the quality of any assignment. Solving a COP thus requires to assign values to the variables such that all the constraints are met and the objective function is minimized. Note that the variables, the constraints and the objective function can be defined at a high-level; the constraints need not be linear or written in SAT clauses for example.

We introduce here after some notations used through this thesis. Let $\mathcal{X} = [X_1, X_2, \ldots, X_n]$ be a set of n variables whose values belong to D, the domain of the variables. We define an assignment as a function $\sigma : \mathcal{X} \to D$ that assigns a value to each variable. We will denote the set of all possible assignments as Λ. A constraint is a function $\mathcal{C} : \Lambda \to \mathbb{N}$ giving the violation of a given assignment. A solution wrt a constraint $\mathcal{C}$ is an assignment σ such that $\mathcal{C}(\sigma) = 0$. An objective is a function $f : \Lambda \to \mathbb{Z}$ giving the evaluation of the quality of a given assignment.

Finally, we define a Combinatorial Optimization Problem (COP) $\mathcal{P}$ as the tuple $\langle f, \mathcal{C}, \mathcal{X}, D \rangle$. Solving a COP requires finding a solution wrt $\mathcal{C}$ minimizing f.

Permutation and partitioning problems are two important classes of COP.

2.1.1 Permutation Problems

A permutation problem on the variables $\mathcal{X} = [X_1, \ldots, X_n]$ with the domain $D = \{1, \ldots, n\}$ asks for an assignment σ of the variables $\mathcal{X}$ such

that the values of all variables are distinct: $\sigma(X_i) \neq \sigma(X_j) \iff i \neq j, \forall i, j = 1, \ldots, n$.

Definition 1. *A permutation constraint on $\mathcal{X}$ is the function $\mathcal{C}_{perm} : \Lambda \to \mathbb{N}$ such that $\mathcal{C}_{perm}(\sigma) = 0$ if and only if σ assigns a permutation of D to the variables $\mathcal{X}$.*

Example 1. *The Traveling Salesman Problem (TSP) is perhaps the best known permutation problem. The TSP calls for a tour of a set of n sites $D = \{1, \ldots, n\}$. A tour can be represented by a permutation of the variables $\mathcal{X} = [X_1, \ldots, X_n]$, where X_i represents which site is visited at the i^{th} position of the tour. Given a distance matrix (c_{ij}), the objective is to minimize the total distance of the tour*

$$f_{TSP}(\sigma) = \sum_{i=1}^{n-1} c_{\sigma(X_i),\sigma(X_{i+1})} + c_{\sigma(X_n),\sigma(X_1)} \tag{2.1}$$

The TSP can thus be represented by the COP $\langle f_{TSP}, \mathcal{C}_{perm}, X, D \rangle$.

2.1.2 Partitioning Problems

In partitioning problems, variables in $\mathcal{X} = [S_0, \ldots, S_{K-1}]$ represent subsets of $D = \{1, \ldots, n\}$.

Definition 2. *A partitioning constraint on $\mathcal{X}$ is a function $\mathcal{C}_{part} : \Lambda \to \mathbb{N}$ such that $C_{part}(\sigma) = 0$ iff σ represents a partition of D, i.e.*

1. $\sigma(S_i) \cap \sigma(S_j) = \emptyset \iff i \neq j$ *with* $i, j = 0, \ldots, K-1$
2. $\bigcup_{i=0}^{K-1} \sigma(S_i) = D$

Example 2. *The Generalized Assignment Problem (GAP) is a partitioning problem. Given a set of tasks $D = \{1, \ldots, n\}$ to be performed, the GAP calls for a partition of D into K machines. The variables $\mathcal{X} = [S_0, \ldots, S_{K-1}]$ represents the partition and S_k represents the set of tasks assigned to machine $k + 1$.*

Each task i has a demand b_i of ressources and the machines have a ressource capacity of B. For each machine k, the sum of the demands of the task assigned to machine k cannot exceed the capacity B: $\sum_{i \in \sigma(S_k)} b_i \leq B$. The violation of this capacity constraint is the following

$$\mathcal{C}_{GAP}(\sigma) = \sum_{k=0}^{K-1} \max\left(0, \sum_{i \in \sigma(S_k)} b_i - B\right) \tag{2.2}$$

There is also a cost c_{ki} to assign task i to machine k and the objective function to be minimized is

$$f_{GAP}(\sigma) = \sum_{k=0}^{K-1} \left(\sum_{i \in \sigma(S_k)} c_{ki} \right) \tag{2.3}$$

The COP $\langle f_{GAP}, \mathcal{C}_{part} + \mathcal{C}_{GAP}, \mathcal{X}, 2^D \rangle$ represents the Generalized Assignment Problem.

2.2 Local Search

Local Search is an effective and intuitive technique for solving combinatorial optimization problems. This technique is also called Neighborhood Search. Any Local Search algorithm relies on a neighborhood function $N : \Lambda \to 2^{\Lambda}$. This function defines, for each assignment, a set of neighbouring assignments. A standard Local Search algorithm starts from an initial solution and improves its quality by iteratively visiting neighbouring solutions.

In many standard Local Search algorithms, the neighborhood of an assignment is defined as the set of assignments that can be obtained by performing slight modifications to it. Such modification of an assignment is called a move; a move is a function $m : \Lambda \to \Lambda$. The set of all possible moves is problem dependent and is denoted $\mathcal{M}$. The neighborhood then becomes $N(\sigma) = \{m(\sigma) | m \in \mathcal{M}\}$.

The feasible neighborhood of an assignment σ wrt a constraint $\mathcal{C}$ is the subset of the neighborhood satisfying the constraint: $N_{\mathcal{C}}(\sigma) = \{m(\sigma) | m \in \mathcal{M} \text{ and } \mathcal{C}(m(\sigma)) = 0\}$. Given an objective function f, a solution σ is a local optimum if no neighbor improves f: $f(\sigma) \leq f(\sigma'), \forall \sigma' \in N_{\mathcal{C}}(\sigma)$.

A simple Local Search algorithm is depicted in Algorithm 1. At each iteration, the algorithm considers the feasible neighborhood of the current solution σ and selects the best candidate wrt the objective function.

2.2.1 Permutation Problems

This section illustrates the concept of Local Search on permutation problems. In the case of the TSP, the move *reverse* proved to be efficient.

A move $reverse(i, j)$ (with $i < j$) represents the operation of reversing the subsequence from position i to position j in the current permutation. There are $O(n^2)$ such possible moves. The differentiation

```
explore(⟨f, C, X, D⟩, N)
    while True do
        Let N_C(σ) = {σ' ∈ N(σ) | C(σ') = 0} ;
        Let N*_C(σ) = {σ' ∈ N_C(σ) | f(σ') < f(σ)};
        if |N*_C(σ)| = 0 then
            break;
        else
            select ( σ' ∈ N*_C(σ) minimizing f(σ') ) do
                Set σ := σ';
```

Algorithm 1: Local Search algorithm solving a COP. A best improvement strategy is used; at each iteration, the algorithm selects the neighbor improving the most objective function f. The search is stopped when there is no improving neighbor.

on the objective function of performing a reverse move $reverse(i, j)$ for the symmetric TSP is

$$\begin{aligned} f(reverse(i,j)(\sigma)) - f(\sigma) = &- c(\sigma(X_{i-1}), \sigma(X_i)) - c(\sigma(X_j), \sigma(X_{j+1})) \\ &+ c(\sigma(X_{i-1}), \sigma(X_j)) + c(\sigma(X_i), \sigma(X_{j+1})) \end{aligned} \tag{2.4}$$

Notice the *reverse* moves preserve the permutation property of an assignment. We can thus define a simple Local Search algorithm by using the following neighborhood

$$N_{reverse}(\sigma) = \{reverse(i,j)(\sigma) | i < j \leq n\}$$

2.2.2 Partitioning problems

This section illustrates the concept of Local Search on partitioning problems. Three moves are usually used in LS approaches. The first move $insert(S_k, i)$ inserts element i in S_k, the move $remove(S_k, i)$ removes element i from S_k and the move $swap(S_k, i, S_m, j)$ removes i and j respectively from S_k and s_m and inserts these two elements respectively in S_m and S_k.

Input	Move	Result
$S_1 = \{1, 2, 3, 4\}$	$insert(S_2, 3)$	$S_2 = \{3, 5, 6, 7, 8, 9\}$
$S_2 = \{5, 6, 7, 8, 9\}$	$remove(S_2, 7)$	$S_2 = \{5, 6, 8, 9\}$
$S_3 = \{10\}$	$swap(S_2, 5, S_1, 2)$	$S_1 = \{1, 3, 4, 5\}$, $S_2 = \{2, 6, 7, 8, 9\}$

Based on these moves, we define two neighborhoods

$$N_{insert}(\sigma) = \{insert(S_k, i) \circ remove(S_m, i)(\sigma) \\ |i \in \sigma(S_m) \text{ and } 0 \leq k \leq K-1 \\ \text{and } 0 \leq m \leq K-1\}$$

$$N_{swap}(\sigma) = \{swap(S_k, i, S_m, j)(\sigma) | i \in \sigma(S_k) \text{ and } j \in \sigma(S_m)\}$$

Notice that the size of these neighborhoods are $O(n.K)$ and $O(n^2)$ respectively, where n is the number of elements in the partition and K is the number of subsets of the partition.

2.2.3 Transition Graph

The concept of transition graph is helpful to better understand the behavior of LS algorithms.

Definition 3. *Given a COP $\mathcal{P} = \langle f, \mathcal{C}, \mathcal{X}, D\rangle$ and a neighborhood function N, the transition graph is the weighted graph $TG = \langle V, E, w\rangle$ such that*

(1) $V = \{\sigma \in \Lambda | \mathcal{C}(\sigma) = 0\}$,

(2) $E = \{(\sigma, \sigma') \in V \times V | \sigma' \in N_{\mathcal{C}}(\sigma)\}$, *and*

(3) $w_{\sigma\sigma'} = f(\sigma') - f(\sigma)$

Local Search algorithms start from a random initial solution $\sigma_0 \in V$ and *walk* along the edges of the transition graph, hoping to reach a good solution at some point. The simplest LS algorithm always selects the most negative edges and restarts the whole process when it reaches a local optimum.

We define the distance $d(\sigma, \sigma')$ between two solutions σ and σ' as the length of the shortest path (considering the number of edges) from σ to σ' in the transition graph. The diameter of a transition graph is the greatest distance among all pairs of nodes.

Two properties of the transition graph are important to characterize the efficiency of a LS algorithm:

1. when **the number of nodes** in the transition graph is big; this allows to walk through the solution space more easily, and

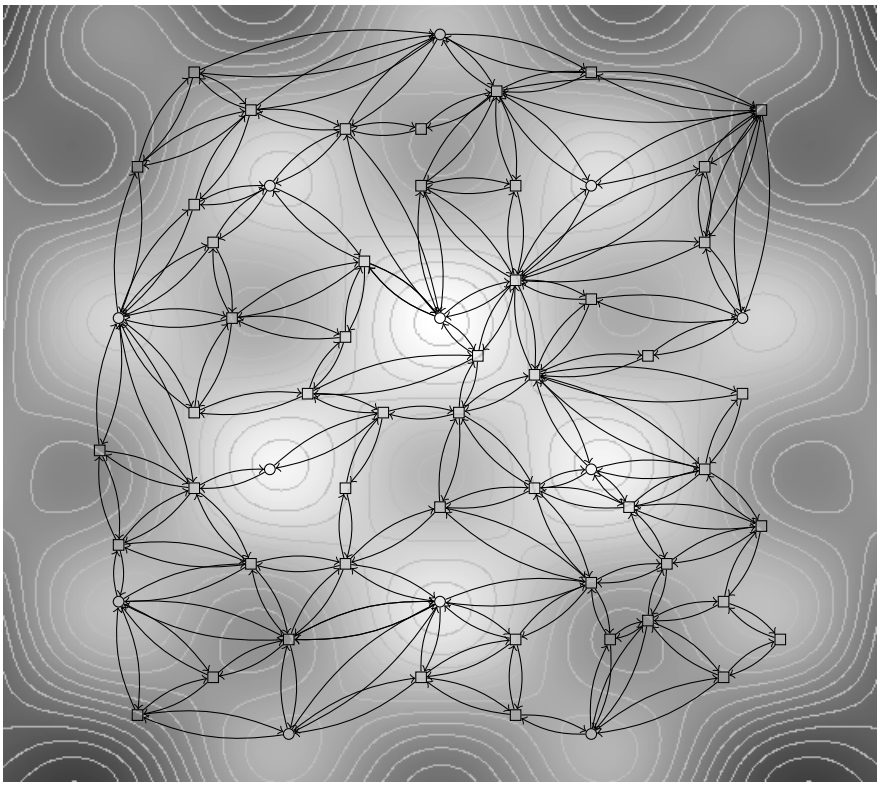


Figure 2.1: The transition graph. Consider the TSP. In this case, the nodes of the transition graph would represent all the permutations of order n. A simple move for this problem is the move $swap(X_i, X_j)$ swapping the values of the variables X_i and X_j. If considering only swap moves, there is an edge (σ, σ') in the transition graph if and only if σ' can be obtained from σ by swapping the values of two variables.

2. when **the diameter** of the transition graph is small; then any node can be reached from any other nodes by traversing a small number of edges

Given the COP $\langle f, \mathcal{C}, \mathcal{X}, D \rangle$, the number of nodes in the transition graph is function of the constraints $\mathcal{C}$; a stronger constraint induces a smaller number of nodes in the transition graph. The diameter of the transition graph is directly function of how much the moves modify the solutions; if the moves in $\mathcal{M}$ only slightly modify a solution, then many moves have to be performed to reach a very different solution.

2.2.4 Metaheuristics

Metaheuristics guide the search when it is stuck in local optima. Given a COP and a neighborhood, many assignments are local optima; there is no neighbor of this assignment that improves the objective function f. A metaheuristic describes how the transition graph should be traversed when the current solution is a local optimum.

Multi-restart

One possible metaheuristic is to perform several runs of a local search algorithm starting from different initial solution.

This metaheuristic can be implemented very easily if the procedure generating initial solutions can be randomized. The advantage of the multi-restart metaheuristic is that it allows to depart from very different initial solutions and then to explore diversified area of the search space. We may say that this metaheuristic performs a very good diversification. However, as soon as a local optimum is reached, the search is restarted. Thus this metaheuristic does not allow to intensify the search near good solutions. This lack of intensification limits the power of this metaheuristic.

Tabu Search

Tabu search allows LS algorithms to explore degrading solutions. This raises the problem of cycling: if the LS algorithm explores a degrading solution σ' when it faces a local optimum σ, at the next iteration, it may well select the local optimum σ again by trying to improve the objective function. The principle behind Tabu Search (TS) is to set the last solutions explored as *taboo* in order not to visit them again. This prevents cycling.

Technically, Tabu search does not keep in memory the entire set of taboo solutions. It only memorizes substructures of taboo solutions. Any solution having the same substructure is considered as *taboo*. Because Tabu search does not memorize entire solutions as taboo, a substructure of solution is set taboo only for a small number of iterations (called tenure). Indeed, a good solution may well share a substructure that is set taboo, and setting this substructure as taboo forever would prevent from visiting this good solution.

Using a Tabu metaheuristic would modify Algorithm 1 by replacing line 4 by

$$\text{Let } N^*_{\mathcal{C}} := \{\sigma' \in N_{\mathcal{C}}(\sigma) | \sigma' \text{ is not tabu}\};$$

Example on the Traveling Salesman Problem Consider the Traveling Salesman Problem. The move $reverse(i, j)$ removes the travels $\sigma(X_{i-1}) \to \sigma(X_i)$ and $\sigma(X_{j-1}) \to \sigma(X_j)$ and replaces them by the travels $\sigma(X_{1-1}) \to \sigma(X_j)$ and $\sigma(X_i) \to \sigma(X_{j+1})$.

Thus, once a move $reverse(i, j)$ is performed, we may decide to force the two new travels to remain in the solution for a few iterations. This may be done by setting the variables $X_{i-1}, X_i, X_j, X_{j+1}$ as taboo for example. This means the value assigned to these variables is not allowed to change in the next iterations.

Advantages and drawbacks Tabu Search tries to escape local minima by exploring solutions close to good solutions in the transition graph. It considers degrading moves and partially prevents the search to cycle in the transition graph. Its implementation is easy and does not require much effort. However this metaheuristic still relies on the neighborhoods structures: if the neighborhoods are inadequate, Tabu Search will perform inefficiently. Moreover, if the neighborhoods used are too small, TS does not diversify the search enough to reach good solutions. Diversification can be added to TS, but requires additional implementation and parameters tuning.

Simulated Annealing

Simulated Annealing (SA) visits more diversified solutions in the search space by relying less on the descent criterion. This metaheuristic accepts to visit degrading neighbors with a probability function of the difference in the objective function wrt the current solution such as illustrated in Algorithm 2. The probability function also depends on the *temperature* that is a parameter decreasing over time. The greater the temperature,

the greater the probability to accept degrading solutions. This temperature allows to largely explore the search space in the beginning of the algorithm, and then to explore more thoroughly promising areas of the search space at the end.

```
1 while True do
2 |  Let N_C(σ) = {σ' ∈ N(σ) | C(σ') = 0} ;
3 |  select ( σ' ∈ N_C(σ) ) do
4 |  |  Δ := f(σ') − f(σ);
5 |  |  Let r ∈ [0; 1];
6 |  |  if Δ < 0 or r < e^(−Δ/t) then
7 |  |  |  Set σ := σ';
8 |  |  update t based on a heuristic;
```

Algorithm 2: Description of a Simulated Annealing algorithm. A feasible neighbor is selected with a probability function of its impact on the objective function.

Advantages and drawbacks The Simulated Annealing metaheuristic diversifies the search by allowing to select *bad moves* at the beginning of the search. The drawback of this metaheuristic is that the temperature may have to be decreased very slowly in order to reach good solutions. This tuning of the temperature update and the time needed to cool it down are the main limitations when using such metaheuristic.

Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a metaheuristic that escapes from local optima by using a collection of neighborhoods $[N^1, \ldots, N^k]$, such as depicted in Algorithm 3. First the neighborhood N^1 is explored. If no neighbor $\sigma' \in N^1(\sigma)$ is found or accepted, then the second neighborhood N^2 is considered, etc. Generally these neighborhoods are increasingly complex to explore; the first ones are fast to find the best neighbor, and if no improving neighbor is found, then we consider worth to explore more complex neighborhoods.

Advantages and drawbacks This metaheuristic allows to escape local minima of one neighborhood by using additional neighborhoods. This metaheuristic is simple to implement, generally because several

```
1 while not stopping criterion do
2 |  for i ∈ 1, ..., k do
3 |  |  Let N_C(σ) := {σ' ∈ N_i(σ) | C(σ') = 0};
4 |  |  select ( σ' ∈ N_C(σ) ) do
5 |  |  |  Set σ := σ';
6 |  |  |  break;
```

Algorithm 3: Variable Neighborhood Search

neighborhoods exist for a given problem. This metaheuristic is also efficient in escaping local minima. However, from empirical experience [HM03], local optima of the different neighborhhoods are close to each other (only a fraction of the variables have different values). Empirically, we also know that the number of variables whose values have to change to escape from a local optimum depends on the quality of this optimum. The best the local optimum the more variables have to take different values to reach a better solution.

These two empirical observations imply that VNS is good at escaping poor local optima. With the size of the neighborhood explored increasing, this metaheuristic will be able to escape from local optima but it is observed that the behavior is then similar to a multi-restart strategy.

Example on the Generalized Assignment Problem We present here a Variable Neighborhood Search metaheuristic for the Generalized Assignment Problem. The VNS uses the insert and the swap neighborhoods introduced in Section 2.2.2. The insert neighborhood is efficient to explore as its complexity is $O(nK)$, and K is generally substancially smaller than n. However, on one instance such that the capacity constraint is almost tight, such as illustrated in Figure 2.2, the insert neighborhood N_{insert} would perform poorly because there is not much free space to move elements.

On the other side, it is more likely for the swap neighborhood N_{swap} to behave very well. Indeed the swap moves do not need much free space to move elements as both elements are first removed and additionnal free space is made. And only then the elements are reinserted. However the complexity of searching the swap neighborhood is $O(n^2)$.

In a VNS metaheuristic, we would thus first search the insert neighborhood and, only if we do not find any improving neighbor, we would search the swap neighborhood.

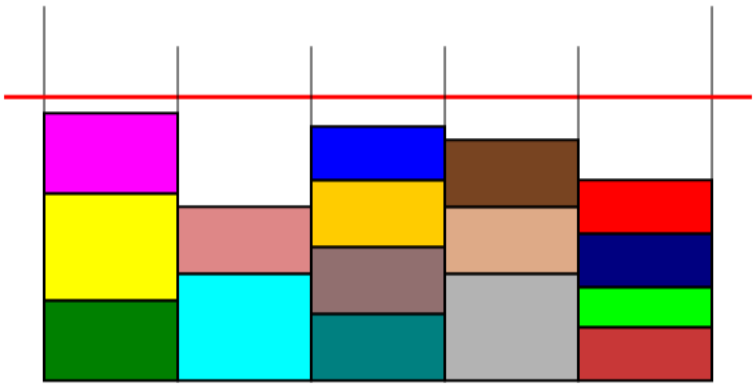


Figure 2.2: Instance of the GAP such that the capacity constraint is tight. The columns represent the machines and the items represent the tasks. The upper line represents the capacity of all machines. In this example, there is not many tasks that can be moved to another machine without violating the capacity constraint.

Accepting unfeasible solutions

Removing a constraint from a COP and adding a violation function of this constraint into the objective is another metaheuristic. When the constraint of the COP to be solved are hard, there are not many feasible neighbors at each iteration of the LS algorithm. One possible solution to overcome this problem is to define a violation function for a constraint $\mathcal{C}_1$ of the COP $\langle f, \mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$. Then we can solve the COP $\langle \alpha\mathcal{C}_1 + f, \mathcal{C}_2, \mathcal{X}, D\rangle$ with α being a high value to favor solutions respecting the constraint $\mathcal{C}_1$. By using such metaheuristic, there exists many more feasible solutions to the new COP, and the traversal of the transition graph is easier. In other words, the transition graph becomes denser, and its traversal is facilitated. The coefficient α can be dynamic in order to explore solutions respecting or not the constraint.

Example on the Generalized Assignment Problem For the GAP $\langle f, \mathcal{C}_{capa} + \mathcal{C}_{part}, \mathcal{X}, D\rangle$, if the capacity constraint is tight, we could not traverse the transition graph efficiently by using the insert neighborhood. However, by considering this constraint as a violation function, the traversal of the transition graph is facilitated. The violation of the capacity constraint is defined as

$$\mathcal{C}_{capa}(\sigma) = \sum_{k=0}^{K-1} \max\left(0, \sum_{i \in \sigma(S_k)} b_i - B\right)$$

We can use the insert neighborhood to solve the COP $\langle f + \alpha.\mathcal{C}_{capa}, \mathcal{C}_{part}, \mathcal{X}, D\rangle$. This COP allows solutions not to respect the capacity constraint, and the transition graph will contains more paths to reach a solution from another.

Advantages and drawbacks This metaheuristic enables to easily accept solutions not respecting a hard constraint and thus to explore the search space more efficiently. However the efficiency of this metaheuristic strongly depends on how the parameter α is updated.

If the parameter α changes too quickly, when the search is driven towards feasible solution, the cost of the solutions explored wrt the objective function becomes poor.

2.2.5 Variable-Depth Neighborhood Search

Variable-Depth Neighborhood Search (VDNS) is a metaheuristic that performs several moves per iteration. Given a set of moves $\mathcal{M}$, we define the set $\mathcal{M}^k(\sigma)$ as the set of solutions that can be obtained from σ by performing at most k moves from $\mathcal{M}$: $\mathcal{M}^k(\sigma) = \{\sigma' \in \Lambda | d(\sigma, \sigma') = k \text{ and } \mathcal{C}(\sigma') = 0\}$. Variable-Depth Neighborhood Search methods heuristically explore $\mathcal{M}^k(\sigma)$ as illustrated in Algorithm 4. This metaheuristic can be considered as heuristically exploring paths of length k from σ in the transition graph. This exploration of $\mathcal{M}^k$ is partial because line 2 only selects one move instead of trying all of them.

Advantages and drawbacks The limitation of this metaheuristic is that $m(\sigma')$ has to be simulated each time a move m is selected, in order to compute $\mathcal{C}(m(\sigma'))$ and $f(m(\sigma'))$ when selecting the next move. This simulation is time-consuming and prevent this metaheuristic to explore many paths. VDNS selects several moves but must simulate each of them in order to be able to compute the effect of the next move to select.

The efficiency of this metaheuristic can be greatly improved if there is an efficient procedure to compute the effect of applying a sequence of moves on a constraint or an objective function. This thesis proposes such procedure. Based on a restriction on the sequence of moves that can be

```
1 Let σ_best := σ;
2 for iter = 1, ..., nbTries do
3 |   Let σ' := σ;
4 |   k := 1;
5 |   while stopping criterion not met or k = K do
6 |   |   Heuristically select m ∈ M|C(m(σ')) = 0 according to
  |   |   f(m(σ'))
7 |   |   |   Set σ' := m(σ');
8 |   |   |   k := k + 1;
9 |   |   if f(σ') < f(σ_best) then Set σ_best := σ';
```

Algorithm 4: VDNS algorithm. This algorithm applies a sequence of up to K moves. These moves are heuristically selected one at a time.

considered, this procedure is able to compute the effect of a sequence of move on a constraint or an objective function.

2.3 Constraint-Based Local Search

The idea of Constraint-based local search [HM05a] is to perform local search on high-level models, enabling the implementation of very efficient and complex LS algorithms for solving complex Combinatorial Optimization Problems. The main added-value of this approach is the ability to quickly validate many different ideas. This is crucial to efficiently solve a new problem. Indeed Combinatorial Optimization is a field such that great ideas have a huge impact on efficiency (far more than code optimization), but has very few tools to predict how an idea will speed-up the Local Search algorithm. By enabling fast prototyping, CBLS helps in finding the best algorithms for solving a problem.

Constraint-Based Local Search (CBLS) expresses Local Search algorithm as two separated components: the model and the search. The model defines the combinatorial optimization problem to solve, i.e. the variables of the problem, the constraints that have to be fulfilled by any solution and the objective function specifying the quality of each solution. The search defines the neighborhood structure ($N(\sigma)$) and how to select a candidate in $N(\sigma)$.

Differentiation is the computation of the impact a move has on the objective function and the constraints. Differentiation is a crucial information to design efficient Local Search heuristics. This has been

illustrated in the metaheuristics presented previously. Moreover Local Search approaches are useful only if a lot of solutions can be explored in short periods of time. Thus the computation of the impact of a move on the objective and constraints has to be very efficient. To do this, state-of-the-art LS implementations use internal data-structures that are maintained incrementally. This leads to implementation requiring complex design, especially when the constraints and the objective functions are not trivial. Constraint-Based Local Search (CBLS) aims at remedying this limitation.

Differential invariants is the key-concept that enables the separation of the model and the search. Differential invariants represent an objective function or a constraint. It provides two main functionalities. First it allows to efficiently differentiate the function they represent wrt different moves, i.e. they enable to compute very efficiently the values

$$\Delta_{\mathcal{C}}(m, \sigma) = \mathcal{C}(m(\sigma)) - \mathcal{C}(\sigma)$$
$$\Delta_f(m, \sigma) = f(m(\sigma)) - f(\sigma)$$

Second, they store and incrementally maintain internal data-structures that allows to compute this differentiation very efficiently. Indeed, these data-structures are crucial to achieve state-of-the-art Local Search algorithms. When a variable is being modified, the differential invariant is *woken up* with the request to update its data-structures accordingly.

The model is defined by using combinations of elementary differential invariants (Figure 2.3). Once a library of elementary differential invariants is available, the objective function and the constraints can be expressed as sums, products, minimum, ... of these invariants. This allows to very quickly modify a model, by adding a constraint as a penalty cost in the objective function for example. The values of one differential invariant can thus depend on the value of another one. This makes very difficult to design Local Search algorithms that updates data-structures and differentiates the model (objective function and consraints). CBLS makes full usage of the model to free the programmer from these difficulties.

All the data-structures of the different invariants can be **updated** efficiently once the model is defined. Notice that the graph induced by the dependencies between invariants (Figure 2.3) is a directed acyclic graph (DAG). Indeed a cycle in the dependency graph would indicate

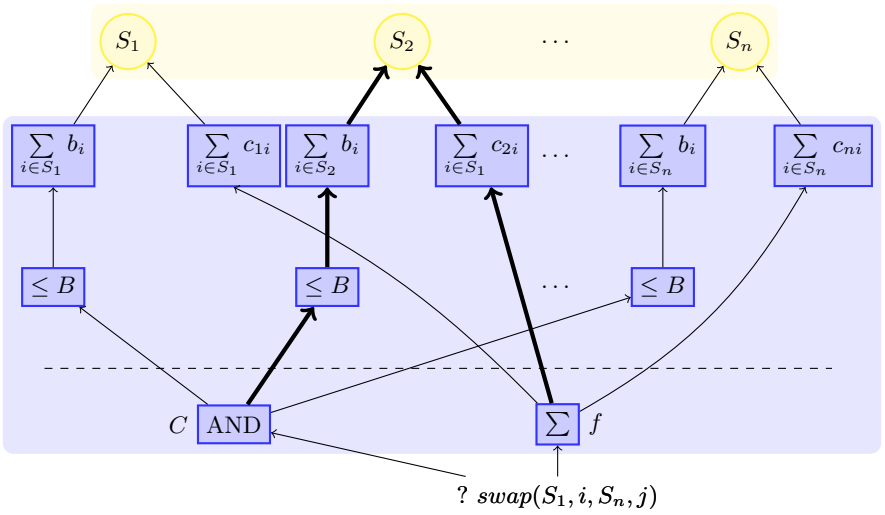


Figure 2.3: Representation of the CBLS model for the GAP. The top level represents all the set variables. The bottom level represents all the differential invariants stating the objective function and the constraints. The sums directly depends on the variables, while the differential invariants representing the global objective function and the constraints depends on other invariants. The thick edges represents the part of the model that has to be updated when the variable S_2 is modified. All differential invariants can be queried to compute their differentiation wrt the move swapping the elements i and j from variables S_1 and S_n for example. This allows to the search procedure to differentiate the model to select the best move to apply eventhough it has no knowledge about the model. This allows the model to be separated from the search procedure, as illustrated by the dashed line.

that the values of the invariants cannot be computed. The model represents a knowledge about the structure of the problem that enable to build the dependency graph. CBLS can then automatically traverse the dependency graph from the modified variable so that it updates an invariant only if all the invariants it depends on have been already updated. This ensures that the values and the data-structures of all differential invariants are consistent.

Constraint-Based Local Search can also efficiently **differentiate** the objective function and the constraints of a model. Indeed the differentiation of a combination of several differential invariants can be computed from the separated differentiations of these invariants. The differential invariant representing the objective function can thus query all the

invariants it depends on to differentiate them and return the global differentiation.

The model thus offers the possibility to be updated wrt modifications to the variables and to be differentiated wrt single moves.

The search of CBLS describes the neighborhood structure and how the neighbors are selected to be visited. The search takes as input the variables, the objective function and the constraints of the model. It has absolutely no knowledge about the structure of the objective function and the constraints. The principle of the search component is to iteratevely apply single move on the solutions. At each iteration, the search component queries the model to get the differentiation of the objective function and the constraints for all the considered moves. It then selects one move based on some metaheuristic and apply it. Once it is applied, the model is auto-updated and the internal data-structures are consistent to differentiate the invariants given the new current solution.

Because the search procedures do not rely on the internal structure of the model, they can be implemented generically. Once a search procedure is implemented, it can be reused on any model. This allows to reuse many complex algorithms and to prototype Local Search algorithms very quickly.

The language COMET implements Constraint-Based Local Search [HM05a]. It enables to very quickly synthetize efficient Local Search algorithms to solve complex Combinatorial Optimization Problems. We here illustrate how the Generalized Assignment Problem can be solved using COMET. The model is depicted in Listing 2.1. It is a strict encoding of the model represented in Figure 2.3. The search procedure is depicted in Listing 2.2. It iteratevely selects the swap move respecting the capacity constraint such it improves the most the objective function. This search procedure does not know anything about the structure of the model. It only has access to the variables and the two differential invariants representing the objective function and the constraints. This illustrates the true separation of the model and the search procedure. This has two main advantages. First it enables to modify the model without the requirement to adapt the search procedure. The model has a great impact on the efficiency of the algorithm. It thus has to be improved very often and the separation of both components allows to prototype these improvement very quickly. Second the separation of components allows to design generic search procedures. This speeds up the initial prototyping phase as existing complex search procedures can

be directly reused.

Comet offers many other tools to help designing complex Local Search or Constraint Progromming algorithms. However they are not required to fully understand the following. They are not covered here. Details can be found in [HM05a].

```
Solver<LS> m();
var{set{int}} S[k in 1..K](m);
ConstraintSystem<LS> C(m);
forall(k in 1..K)
  C.post( sumOver(S[k],b) <= B );
Function<LS> f =
      sum(k in 1..K) sumOver(S[k],all(i in 1..n) c[k,i])
       ;
```

Listing 2.1: Comet implementation of the model of the GAP illustrated in Figure 2.3.

```
while( true ){
  selectMin(k in 1..n, l in k+1..n, i in S[k], j in S[l]
              : C.getSwapDelta(S[k],i,S[l],j)==0,
              d = f.getSwapDelta(S[k],i,S[l],j))
           (d){
   if (d >= 0) break;
     swap(S[k],i,S[l],j);
  }
}
```

Listing 2.2: Comet implementation of a best improvement search procedure to solve partitioning problems. The call `C.getSwapDelta` returns the value $\Delta_{\mathcal{C}}(m, \sigma)$, which is the impact of applying the corresponding move on the violations of the constraints. The same holds for the call `f.getSwapDelta`. This procedure thus select the swap moves that respects the constraints and that minimizes the objective function. The call `swap(S[k],i,S[l],j)` applies the move.

Chapter 3

State-of-the-art of VLSN

This chapter presents the existing works on VLSN. First it defines what is a VLSN, then illustrates how VLSN can be defined as the compound of several atomic moves. Finally this chapter presents VLSN that are defined by relaxations of the problem to solve. Because these ideas have been used to solve several applications in many cases, the following sections first describe an approach, then enumerate the related papers.

3.1 Very Large-Scale Neighborhoods

A Very Large-Scale Neighborhood (VLSN) is a neighborhood containing a huge numbers of neighbouring assignments.

Definition 4. *Given a COP $\langle f, \mathcal{C}, \mathcal{X}, D \rangle$, a Very Large-Scale Neighborhood is a neighborhood $N : \Lambda \rightarrow 2^{\Lambda}$ respecting the two following properties:*

1. *the number of neighbors to any solution σ of the problem is very high, generally exponential ($|N(\sigma)| \geq O(2^n)$) or polynomial with a high degree ($|N(\sigma)| \geq O(n^3)$), where n is the size of the problem.*

2. *there exists a very efficient algorithm to compute a good, or the best neighbor $\sigma' \in N(\sigma)$ wrt the objective function f.*

So the main idea behind VLSN is to create an *algorithmic leverage* between the size of the neighborhood and the time-complexity required to identify the next neighbor to consider. For example, an algorithm in $O(n^2 \log n)$ could return the best neighbor from a neighborhood whose size is in $O(2^n)$. Examples will be given in the following sections.

There exist two main approaches to design exponential-size VLSN. In the first one, a VLSN is defined as the set of solutions that can be

obtained by applying several atomic moves. In the second, the current solution is relaxed and then reoptimized. These two approaches are reviewed in the two following sections.

3.2 Compounding Moves

In many standard local search algorithms, only one move $m \in \mathcal{M}$ is selected at each iteration: given a COP $\mathcal{P}$, if $LS(\mathcal{P}, \sigma)$ denotes the neighborhood used in standard local search approaches, $|LS(\mathcal{P}, \sigma)| = O(|\mathcal{M}|)$. In such approaches, a single move cannot violate the constraints of the problem. Then in the case of problems much constrained, few moves are feasible and the size of the feasible neighborhood $N_{\mathcal{C}}(\sigma)$ is very small. This is a limitation of standard local search approaches.

The aim in designing VLSN by compounding moves [EOSF02] is to remedy this limitation by applying several moves per iteration:

$$VLSN(\mathcal{P}, \sigma) = \{m_1 \circ \cdots \circ m_k(\sigma) | [m_1, \ldots, m_k] \text{ is a sequence of moves in } \mathcal{M}\}$$

The size of the neighborhood is exponential ($O(2^{|\mathcal{M}|})$). Moreover a single move can be represented by a sequence containing only this move, thus the neighborhoods used in VLSN approaches are supersets of the neighborhoods used in standard local search approaches ($LS(\mathcal{P}, \sigma) \subseteq VLSN(\mathcal{P}, \sigma)$).
This type of VLSN has two great advantages compared to standard local search. First, the diameter of the transition graph is reduced (Section 3.2.1); by applying several moves, many variables are modified per iteration, and less iterations are thus needed to go from one solution to another.

Second, the transition graph is often densified (Section 3.2.2); because we apply several moves, we may consider atomic moves violating some constraints, the challenge being to select moves such that the overall effect of applying them does not lead to the violation of any constaint.

3.2.1 Decreasing the diameter of the transition graph

One effect of using VLSN by compounding moves is the reduction of the diameter of the transition graph, compared to standard local search neighborhoods. We illustrate it here by describing the DynaSearch VLSN on the Traveling Salesman Problem [PvdV95].

A VLSN has been proposed in [PvdV95] for solving the TSP, where a neighbouring solution is obtained by applying the conjunction of several indenpendent reverse moves.

A move $reverse(i, j)$ (with $i < j$) represents the operation of reversing the subsequence from position i to position j in the current permutation. There are $O(n^2)$ such possible moves. The differentiation on the objective function of performing a reverse move $reverse(i, j)$ for the symmetric TSP is

$$\begin{aligned}\Delta_{f_{TSP}}(reverse(i,j),\sigma) = &- c(\sigma(X_{i-1}),\sigma(X_i)) - c(\sigma(X_j),\sigma(X_{j+1})) \\ &+ c(\sigma(X_{i-1}),\sigma(X_j)) + c(\sigma(X_i),\sigma(X_{j+1}))\end{aligned} \tag{3.1}$$

In the context of the TSP, two moves $reverse(k, l)$ and $reverse(i, j)$ are *independent* iff $l < i - 1$ or $j < k - 1$. Let two independent reverse moves m_1 and m_2. From (3.1), the variation on the objective function of applying m_2 is independent whether m_1 is applied or not:

$$\Delta_{f_{TSP}}([m_1, m_2], \sigma) = \Delta_{f_{TSP}}(m_1, \sigma) + \Delta_{f_{TSP}}(m_2, \sigma)$$

This allows to compute the cost of a reverse move *a priori*, then to select several independent moves and apply them. The total variation on the objective function of applying these moves is exactly the sum of the single variations of the moves considered separately.

To compute the best neighbor of this VLSN, we use an *improvement graph* that is a directed graph $G(\sigma) = (V, E, w)$, where $V = \{1, 2, \ldots, n, 1', 2', \ldots, n'\}$; there are two nodes i and i' per position in the tour and $E = \{(i, j') : i < j\} \cup \{(j', k) : k > j + 1\}\}$. An edge (i, j') represents the move $reverse(i, j)$ with the associated cost $\Delta_{f_{TSP}}(reverse(i, j), \sigma)$ and an edge (j', k) does not represent any move (its cost is 0). We also add an edge from a dummy source node S to any node i and an edge from any node j' to a dummy sink node T. We look for a path from S to T (Figure 3.1). The effect on the permutation and the cost of each edge selected is depicted in Figure 3.2.

The improvement graph is acyclic. The shortest path can thus be computed in linear time wrt the number of edges (i.e. wrt the number of reverse moves). Searching the DynaSearch neighborhood has thus the same time complexity as searching the simple reverse neighborhood, even if the size of the DynaSearch is much larger.

Because many reverse moves are applied at each iteration, the transition graph is traversed more quickly. Indeed, we need less iterations to pass from a given solution to another. The diameter of the transition

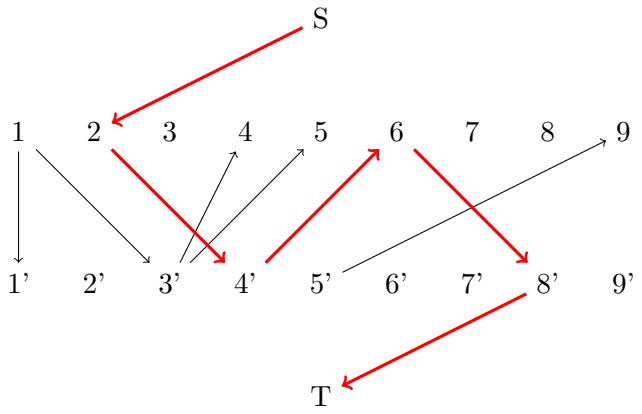


Figure 3.1: The improvement graph for the independent reverse move. Any *downward* edge represents a reverse move. Any upward edge does not represent any specific move. Any path from S to T represents a set of independent reverse moves: applying them yields a variation on the TSP objective function that is equal to the sum of the cost of the edges of the path. This improvement graph is thus specific to the Traveling Salesman Problem.

graph of a VLSN is thus reduced wrt standard local search neighborhoods. The same reasoning can be done when using other simple moves such as swap or insert moves, in place of reverse moves.

Bibliography The Dynasearch methodology has been applied to several permutation problems. For all these works, the atomic moves are feasible wrt all the constraints of the problem. Dynasearch has been applied to the TSP in [PvdV95, Con00], to the single-machine total weighted tardiness scheduling problem in [CPvdV02, EO06b, GCT04], to the one-batching machine problem in [Hur99], and to the linear ordering problem in [Con00].

3.2.2 Densifying the Transition Graph

The second effect of using VLSN is the densification of the transition graph, compared to standard Local Search. Indeed, standard local search approaches only consider moves that respect all the constraints of the problem. VLSN densifies the transition graph by considering atomic moves that separetely violate some of the constraints. However, the set

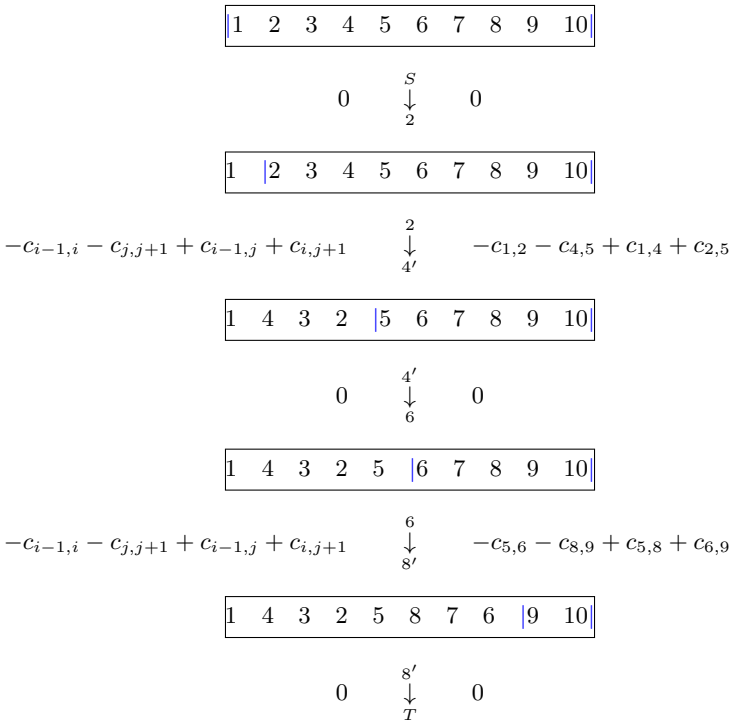


Figure 3.2: Illustration of the effect of the different independent reverse moves represented by the path in Figure 3.1. The general formula of the cost of the edges is represented on the left. The specific cost of the edges of this path is given on the right.

$M \subseteq \mathcal{M}$ of atomic moves to apply is selected such that these constraints are respected after the application of all the moves in M.

Cyclic exchanges for partitioning problems

We illustrate such VLSN here on Generalized Assignment Problem described in Section 2.

The swap move is a standard move for partitioning problems. Such moves preserve the partitioning structure of an assignment. The move $swap(S_k, i, S_l, j)$ consists in swapping the elements i and j among the two subsets S_k and S_l. For example if $\sigma(S_1) = \{1, 2, 3, 4\}, \sigma(S_2) = \{5, 6, 7\}$ then applying the move $swap(S_1, 1, S_2, 5)$ leads to the assignment $\sigma'(S_1) = \{2, 3, 4, 5\}, \sigma'(S_2) = \{1, 6, 7\}$. A local search algorithm only considering swap moves only visits solutions that respects the partitioning structure.

Using exchange moves allows to consider more solutions and thus densifies the transition graph. Indeed the exchange move violates the partitioning structure of the assignments. The move $exchange(S_{\sigma_j}, i, j)$ inserts i in S_{σ_j} [1] and removes j from this same subset. For example if $\sigma(S_1) = \{1, 2, 3, 4\}, \sigma(S_2) = \{5, 6, 7\}$ then applying the move $exchange(S_2, 1, 5)$ leads to the assignment $\sigma'(S_1) = \{1, 2, 3, 4\}, \sigma'(S_2) = \{1, 6, 7\}$. Clearly exchange moves do not respect the partitioning structure of an assignment, and we can consider more assignments than an when using swap moves. Indeed the swap move $swap(S_{\sigma_i}, i, S_{\sigma_j}, j)$ is equivalent to the moves $exchange(S_{\sigma_j}, i, j)$ and $exchange(S_{\sigma_i}, j, i)$.

Several exchange moves are selected and applied in order to respect the partitioning constraint. A cyclic exchange $[e_1, e_2, \ldots, e_m]$ is equivalent to the moves $\{exchange(S_{\sigma_{e_2}}, e_1, e_2), \ldots, exchange(S_{\sigma_{e_1}}, e_m, e_1)\}$ (Figure 3.3). The exchange moves separately violate the partitioning constraint but applying a cyclic exchange preserves the partitioning structure of an assignment.

Cyclic exchanges do not allow to change the cardinality of the subsets of the partition. A path exchange $[e_1, e_2, \ldots, e_m]$ is equivalent to the moves $\{remove(S_{\sigma_{e_1}}, e_1), exchange(S_{\sigma_{e_2}}, e_1 e_2), \ldots, insert(S_{\sigma_{e_m}}, e_{m-1})\}$.

The cyclic neighborhood of an assignment σ is the set of assignments that can be obtained from σ by applying path or cyclic exchanges. Whereas there are $O(n^2)$ 2-exchanges, there exists an exponential number $O(n^K)$ of cyclic-exchanges. The cyclic neighborhood is thus a VLSN.

[1] Given a solution σ, the index of the variable containing the element i is denoted σ_i: $\sigma_i = k$ if and only if $i \in \sigma(S_k)$.

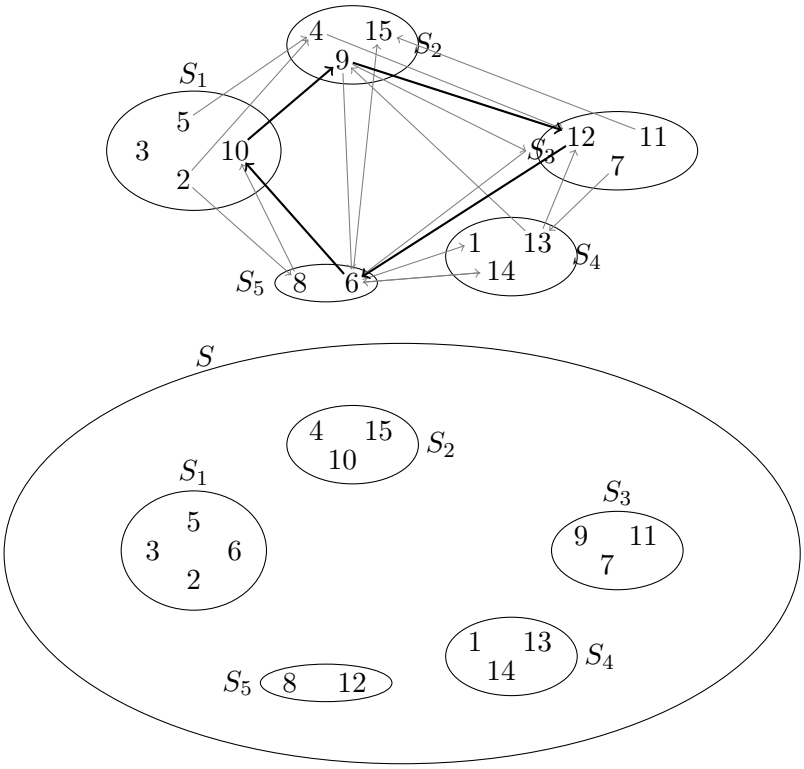


Figure 3.3: Cyclic exchanges for partitioning problem. The first picture represents a solution to the partitioning problem, a cyclic exchange, and the associated improvement graph.The second picture illustrates the partition after the application of the cyclic exchange corresponding to the color-disjoint cycle $[10, 9, 12, 6]$.

A dedicated improvement graph is used to compute a cyclic or path exchange respecting the constraint (Figure 3.3). The improvement graph of an assignment σ is a weighted graph $IG(\sigma) = (V, E, w)$ where $V = V_1 \cup V_2$, $V_1 = \{1, \ldots, n\}$, $V_2 = \{S_1, \ldots, S_K\}$ and $E \subseteq V_1 \times V_1 \cup V_1 \times V_2 \cup V_2 \times V_1$. An edge $e = (i, j) \in V_1 \times V_1$ represents the move $exchange(S_{\sigma_j}, i, j)$. The difference in cost of performing this operation is associated to the edge (i, j): $w_e = \Delta_f(exchange(S_{\sigma_j}, i, j), \sigma) = -c_{i,\sigma_i} + c_{i,\sigma_j}$. Such edge $e \in E$ if and only if the operation of exchanging j by i in the current solution does not violate the capacity constraint: $(i, j) \in E \iff \Delta_{\mathcal{C}_{GAP}}(exchange(S_{\sigma_j}, i, j), \sigma) = 0 \iff \sum_{e \in S_{\sigma_j}} b_e - b_j + b_i \leq B$. An edge $e = (i, S_k) \in V_1 \times V_2$ represents the move $insert(S_k, i)$, has an associated cost $w_e = c_{i,k}$ and $e \in E$ if and only if inserting the element i in S_k does not violate the capacity constraint. An edge $e = (S_k, i) \in V_2 \times V_1$ represents the move $remove(S_{\sigma_i}, i)$, has an associated cost $w_e = -c_{i,\sigma_i}$ and is always in E, because removing an element always respect the capacity constraint.

The costs and the presence of the edges in the improvement graph were computed by considering the corresponding moves separately. We now look at what happens to the objective function and the constraints when we apply all the moves corresponding to a cycle in the improvement graph.

The costs assigned to the edges reflect the variation of the objective function when performing the corresponding moves. Because the objective function is a sum of independent terms, the variation on the objective function of performing several moves is exactly the sum of the costs assigned to the edges of the cycle.

Clearly there is a one-to-one correspondence between cycles in the improvement graph and cyclic or path exchanges. Thus applying the moves assigned to a cycle in the improvement graph will respect the partitioning constraint.

However this is not true for the capacity constraint. When we check whether a move respects the capacity constraint (modifying the subset S_k), we compute the total demand of the elements in S_k: $\sum_{r \in S_{\sigma_j}} -b_j + b_i \leq B$. This implicitely assume that no other move will modify the variable S_k. We must then find cycles containing moves that do not modify common set variables. This is done by assigning a different color to the K subsets and coloring the edges of the improvement graph according to the variable modified by the corresponding move. Then we search for color-disjoint cycles [TO89] (Figure 3.3). This problem is NP-Hard, but efficient polynomial heuristics exist.

Bibliography These cyclic exchange VLSN for partitioning problems and their associated improvement graph were first introduced in [Tho88, TO89, TP93]. This technique has been used on a wide range of problems from combinatorial optimization and is similar to the one described above for the GAP. They present a heuristic to find subset-disjoint cycle with negative cost. Their algorithm will be presented later in Section 4.3. Here below we list several applications solved by considering cyclic exchanges.

Transportation Problems The Vehicle Routing has been solved several times by means of this VLSN, see [FW90, GGPS06, IIK$^+$05, TP93, CFS$^+$04, CDS08]. The Combined Through And Fleet Assignment Problem has been tackled in [AGM$^+$01]. A real-life locomotive scheduling problem has been solved in [ALO$^+$02]. The Single Source Transportation Problem is solved with the Covering Assignment Problem in [ÖKNP08] by a two stage improvement procedure. First, for each subset of the partition, they select a subset of elements to exchange. Then they use a matching algorithm to cyclicaly exchange these subsets among the subsets of the partition. Their neighborhood captures the cyclic neighborhood, however they must first select the element to cyclically exchange. This increases the heuristic aspect of the search algorithm.

Distribution Problems The Single Source Capacitated Facility Location Problem is treated in [AOP$^+$02]. The authors in [AOS01, AOS03] obtained the best known solution for the capacitated minimum spanning tree problem, having interest in telecommunication network design. The K -Constraint Multiple Knapsack Problem [AC05], the weapon target assignment problem [AKJO03] have also been tackled by VLSN.

Scheduling Problems This technique has also been applied to examination timetabling problem [AABD04, AABD07, ADSV06, MO07], the Generalised Assignment Problem is solved by VLSN in [Dus02, YIIG04]. The minimum makespan machine scheduling problem is tackled by VLSN in [FNS00]. VLSN have also been used on the one-machine batching problem [Hur99], Integrated Clustering and Machine Setup Model for PCB Manufacturing [MPS02].

Graph Colouring Problem A different approach is presented in [GPB05] for solving the Graph Colouring Problem. They partition the nodes in two subsets and perform pairwise permutations of colors into

one of these sets. The best neighbor is computed by solving a matching problem.

Cyclic exchanges for permutation problems

The cyclic-exchange is defined quite similarly for permutations and partitions [AJOS02]. An exchange move $exchange(i, j)$ is defined as removing the value j from its current position in the permutation and replacing it with the value i. A cyclic-exchange $[x_1, x_2, \ldots, x_k]$ is defined as the conjunction of k exchange moves $exchange(x_1, x_2), \ldots, exchange(x_k, x_1)$. Clearly one single exchange move does not preserve permutations, however cyclic-exchanges do.

In order to compute the best neighbor of this cyclic-exchange neighborhood, we build an improvement graph $G(\sigma) = (V, E, w)$ similarly as for partition problems. V is the set of values in the sequence to be permuted and an edge $(i, j) \in E$ represents the move $exchange(i, j)$. The variation of this move on the objective function is associated to the edge. In order to be able to compute these costs *a priori* we restrict cyclic-exchanges to only contain independent moves. This notion of independance is problem specific. We illustrate it on the TSP.

For the TSP, the cost of the move $exchange(i, j)$ is equal to

$$w_{ij} = -c_{j^-,j} - c_{j,j^+} + c_{j^-,i} + c_{i,j^+}$$

where j^- refers to the city visited just before city j and j^+ the city visited right after j. This is illustrated in Figure 3.4. Clearly this cost calculation assumes that the cities j^- and j^+ will remain at the same position. So two exchange moves $exchange(k, l)$ and $exchange(i, j)$ are *independent* iff $j \neq l^-, j \neq l^+, l \neq j^-, l \neq j^+$.

Once these cost have been calculated, any cycle containing only independent moves represents a cyclic-exchange move and the cost of this cycle is exactly the difference in cost of the permutation observed by performing this move.

Bibliography The quadratic assignment problem has been solved by this approach in [AJOS02]. In [BS01], the TSP is solved with cyclic exchange with the permutation being represented by the successor variables. In this work, the best candidate is computed by a dynamic program.

The pyramidal neighborhood defined in [CV90] can also be conceived as the compound of several atomic moves that break the permutation constraint. Here the set of moves to apply in order to respect the

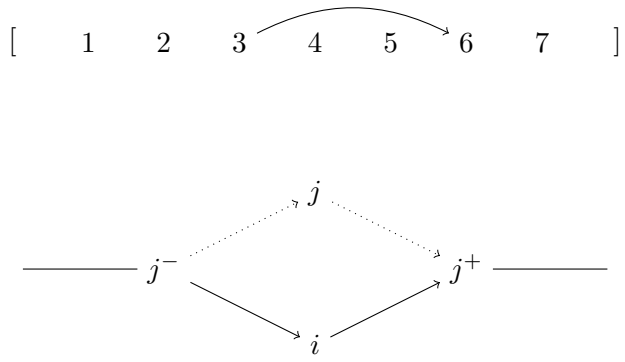


Figure 3.4: The exchange move for the TSP. The move $exchange(3, 6)$ replaces 6 by 3 in the current permutation. This move implies that city j won't be visited between cities j^- and j^+; instead i will be visited in this position. Thus the cost associated to this move is due to the removal of arcs (j^-, j) and (j, j^+) and the insertion of arcs (j^-, i) and (i, j^+). The cost of this move is $w_{ij} = -c_{j^-,j} - c_{j,j^+} + c_{j^-,i} + c_{i,j^+}$.

permutation constraint is computed by means of a shortest path algorithm. This is also the case for the neighborhoods proposed in [BS01] and [BD95].

Matching-based VLSN

The ASSIGN neighborhood defined in [SD81] for solving the TSP is also a nice example of VLSN that can be searched exactly. The algorithm considers the current tour and eject k pairwise non-adjacent cities. The neighborhood is formed from all tours that can be obtained from the current tour by permuting these k cities, i.e. reinserting them in the positions left free. To find the best neighbor, a bipartite graph $G(\sigma) = (V_1, V_2, E)$ is built with V_1 representing the k cities removed and V_2 the k positions of these k cities. An edge (i, j) represents the move of reinserting the city i to the position j and the associated change in the objective function is associated to the edge. Then clearly we need to find a set of edges $E' \subseteq E$ such that exactly one edge in E' is adjacent to each city in V_1 and to each position in V_2. In order to find the best improving neighbor we can thus solve a maximum matching problem on $G(\sigma)$.

Bibliography This neighborhood has been first presented in [SD81]. Then several variants of this neighborhood were published. In [Pun01]

subtours are permuted instead of single cities. Other low complexity algorithms for solving similar neighborhoods are given in [Gut99] and some theoretical results on these neighborhoods are presented in [GY99]. Other variants are given in [DW00].

Larger neighborhoods based on the ASSIGN neighborhood and that are not exactly searchable in polynomial time are illustrated in [Kar79, GGYZ01, PK02]. The Inventory Routing Problem is discussed in [DL86]. A neighborhood similar to the ASSIGN neighborhood is used for solving the Car Sequencing Problem in [EGN06]. They use an IP solver to find an assignment meeting additional side-constraints.

Some other papers [DL86, Tai03, Tai93] discuss a neighborhood for the Vehicle Routing Problem where subsets of a partition are merged two by two. A matching algorithm is used to find the best pairwise-merges to perform in order to decrease the objective value.

3.2.3 Searching the neighborhood heuristically

In some algorithms, the subset M' of moves to be performed at once is built by adding one move from a set M at a time. This approach is generally called Variable Depth Neighborhood Search (VDNS).

The k-distance neighborhood of the assignment σ is defined as $\{\sigma' \in \Lambda : d(\sigma, \sigma') \leq k\}$. The main idea behind VDNS is to partially explore the k-distance neighborhood obtained by performing a sequence of atomic moves on the current solution σ. If δ is the maximum degree of the nodes in the transition graph, then the size of the k-distance neighborhood is $O(\delta^k)$. Searching this neighborhood entirely is generaly too time-consuming for values of k greater than 3. Searching this neighborhood can be seen as exploring a search tree T whose set of nodes is the k-distance neighborhood, whose root is the current solution σ and where an edge is in T if its endpoints are two adjacent solutions in the transition graph. The goal is to find a path from σ to any other node σ' in T, hopefully with a lower objective value.

In order not to explore all the nodes in T, the user defines a stochastic function $MoveVDNS : \Lambda \rightarrow N(\sigma)$. Then T is explored by using $MoveVDNS$ to generate several paths from σ to another solution σ' in the k-exchange neighborhood. The solution σ' of lowest cost is selected as the next solution visited by the algorithm.

The most well-known VDNS is the Lin-Kernighan heuristic [LK73] for the Traveling Salesman Problem(TSP).

One large part of the VDNS algorithms can be considered as ejection chains algorithms [GR06]. In this kind of algorithm, we alternate by first

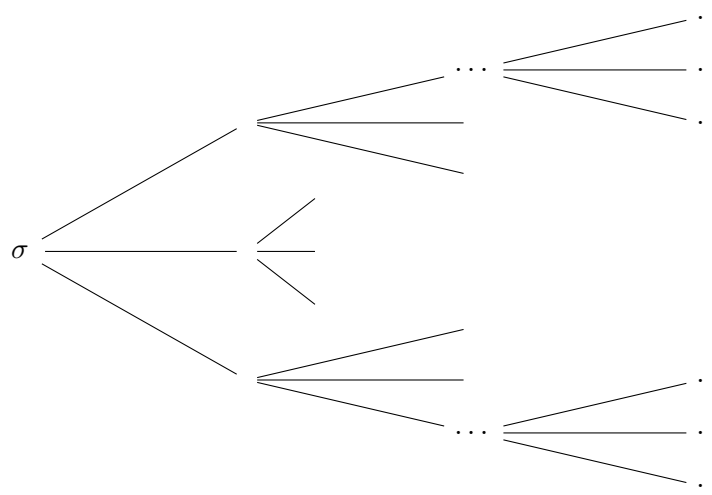


Figure 3.5: Variable-depth Neighborhood Search. Here the procedure $MoveVDNS$ is used three times at each nodes, to generate three neighbors. The size of the tree is thus 3^k.

removing one element from the current solution and then reinserting it differently. The Lin-Kernighan heuristic can be considered as an ejection chains based algorithm. In [GR06], a Filter and Fan methodology is presented. This approach can also be considered as a VDNS algorithm.

See [AOEOP02] for a comprehensive list of papers using VDNS to solve optimization problems.

3.2.4 Abstracting Compound VLSN

Some work towards unification of these exponential neighborhoods has been performed. They mainly try to derive the search algorithm from the structure of the VLSN. Unfortunately, none of them lead to experimental results comparable to the dedicated algorithms described previously.

In [BDW98, DW00], the authors use permutation trees to represent a set containing an exponential number of permutations at each iteration of the local search algorithm. The best improving permutation in this set can be identified in polynomial time when the permutation tree has a special structure by means of a dynamic programming approach.

Theoretical work about VLSN and the TSP such as the analysis of the diameter of the transition graph can be found in [GG05, GP02, GYZ04].

In [EO06a] a methodology is presented in the same spirit of deriving dynamic programming algorithms for searching exponential neighbor-

hoods for the TSP. The authors consider the generic Held and Karp formulation of the TSP [HK61] and restrict this DP to a polynomial number of states only. Solving this DP can be done in polynomial time depending on the states chosen for the restriction. Their approach unifies the VLSN presented in [BS01, CV90, PvdV95]. Following this work, a methodology using grammars to define a VLSN has been presented in [BO05]. The neighborhood is defined as the set of permutation generated by a grammar given by the user and a generic DP can find the best improving permutation in polynomial time depending on the grammar size. This generic DP achieves the same theoretical time complexity as dedicated algorithms for the most well-known VLSN on the TSP. No experimental results were given. These approaches allow to quickly define a VLSN for permutation problems and use a generic algorithm to search it. However, they operate at a low level of abstraction and the VLSN has to be cleverly designed in order to ensure that all the neighbors satisfy the constraint and have the expected cost; the VLSN defined with such approach cannot self-adapt in function of the model.

Recent work relies on the variables to select moves that may be applied together [Ben10] with the ASSIGN neighborhood. They model combinatorial problems with boolean variables. Constraints are expressed as equality between sums of such variables. They consider moves flipping the values of several boolean variables. This approach has two main limitations. First, they allow to select two moves together only if they do not modify variables that are in the scope of a common constraint. This disallows the use of global constraints, a cornerstone of constraint-based approaches, because the scope of such constraint is generally a large subset of the decision variables. Second, they assume that a given move always modifies the same variables. This restricts their approach to moves that always modify a small fixed subset of variables. The moves presented in Section 3.2.2 do not satisfy these severe restrictions.

3.3 Solvable special cases of the application

This section reviews some VLSN designed from special cases of the original problem to solve. Typically these cases are built by restricting the topology or by adding additional constraints to the original problem. In local search algorithm, the neighborhood considered can be the set of solutions to these special cases. We usually consider special cases having an exponential number of solutions. The special cases can either be solved by a dedicated polynomial-time algorithm or by Branch-and-

Bound algorithms (dedicated or based on Constraint Programming (CP) or Integer Programming (IP)).

This section is here for information only. Because our work does not extend any appraoch presented in the following, this section does not aim at being exhaustive.

3.3.1 Neighborhoods created by relaxing the current solution

In this section we present neighborhoods obtained by considering the current solution, relaxing it, either by relaxing the values of some variables, or by relaxing some constraints, and then using a complete search algorithm to find the best solution of this relaxation, optimizing the objective function. Usually these search algorithm are called Large Neighborhood Search and a CP or IP solver is used to find the best solution.

This approach is generally useful when one wants to define a neighborhood respecting many side constraints.

Large Neighborhood Search is introduced in [Sha98] for solving the Vehicle Routing Problem. Given a solution, some visits are removed from the current route and a CP solver is used to reinsert them optimally. Further work used the same approach to consider additional side constraints to the vehicle routing problem: time windows [BH04] and pickup and delivery [BH06]. The reallocation can also be optimized by Integer Programming [FFT05].

In [ALO+02], a similar approach is proposed for solving the Locomotive Scheduling Problem: assign a set of locomotives to each train in a pre-planned train schedule. At each iteration, the algorithm relaxes the trains assigned to only one locomotive type and reassigns all the locomotives of this type using a IP solver.

3.3.2 Neighborhoods created from a special case of the application

For all problems in combinatorial optimization, if there exists a special restriction solvable in polynomial time, one can derive a VLSN by designing an algorithm that given a solution σ to the problem generate a restriction of the original problem such that σ is a solution of this special case. The polynomial-time algorithm is then used to find the best improving neighbor.

One such example for the TSP is given in [CNP83]. At each iteration, they add edges to the current tour in order to create a Halin graph (a tree plus edges to form a cycle around all the leaves). They optimize

the TSP in polynomial time on this graph to find the best improving neighbor.

More special cases of the TSP were presented in [Yeo97, BD95, GP97].

Many special such examples exist for the TSP and other problems and each of them could lead to a new exponential neighborhood.

Part II

Contributions

Chapter 4

Constraint-Based Very Large-Scale Neighborhood Search

This chapter presents some theoretical abstractions of the VLSN presented in the previous chapter. These abstractions will be useful to implement Constraint-Based Very Large-Scale Neighborhood (CBVLSN) Search that is a framework for building efficient VLSN search algorithms in the next chapter.

4.1 A Theory of Constraint-Based VLSN Search

The last section illustrated how selecting several moves at each iteration enables to consider moves violating some constraints, and thus densifying the transition graph. The VLSN presented are the most successful in the literature, in terms of applicability on hard real-life problems. However, such VLSN are dedicated to a particular problem, from the definition of the moves to the definition of the improvement graph. One goal of this thesis is to abstract the concepts presented here in order to create modular and reusable VLSN components.

In VLSN search, neighbors of a solution are reached by applying a sequence of moves and the size of the neighborhood is exponential in the length of the sequence. In general, the moves considered violate a given global constraint. However the sequence of moves is selected such that this global constraint is respected after the application of all the moves in the sequence. This approach raises two fundamental problems: (1) how to select a sequence of moves such that a global constraint is not violated, and (2) how to compute efficiently the variation of a sequence

of moves on the objective function and the constraints

These problems are solved by considering only sequences of moves respecting some properties. These properties are stated by the concept of MoveGraph to deal with the first problem, and by the concepts of independence and compositionality for the second.

4.1.1 Applying several moves

This research is about designing tools to efficiently compute the effect of applying a sequence of moves on an assignment. To achieve this goal, it is important to know exactly how a single move of this sequence will modify the current assignment. We thus define the sequential application of several moves on an assignment. Intuitivelly, in the sequential application of moves m_1 and m_2 of an assignation σ, both moves m_1 and m_2 are applied on the *initial* assignation σ. If the resulting two assignations diverges on the value of a variable, the value resulting from move m1 is chosen for this variable.

Definition 5. *Given an assignment σ and two moves m_1 and m_2 the sequential application $m_1|m_2$ of these two moves wrt σ is such that*

$$m_1|m_2(\sigma)(X_i) = \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{otherwise} \end{cases}$$

Example 3. *Consider assignments on variables X_1 and X_2. We define two moves m_1 and m_2 such that $m_1(\sigma)(X_1) = \sigma(X_1) + 1$, $m_1(\sigma)(X_2) = \sigma(X_2)$, $m_2(\sigma)(X_1) = \sigma(X_1)$ and $m_2(\sigma)(X_2) = \sigma(X_1)$.*

Let us take $\sigma = (2, 1)$. We have $m_1(\sigma) = (3, 1)$, $m_2(\sigma) = (2, 2)$ and $m_1|m_2(\sigma) = (3, 2)$. Note that we also have $m_2|m_1(\sigma) = (3, 2)$.

Proposition 1. *The sequential application is associative.*

Proof. Given any assignment σ, we prove that $(m_1|m_2)|m_3(\sigma)(X_i) = m_1|(m_2|m_3)(\sigma)(X_i)$ for all variable X_i.

Let σ be any assignment, X_i be any variable and m_1, m_2, m_3 be any three moves. We have

$$\begin{aligned} (m_1|m_2)|m_3(\sigma)(X_i) &= \begin{cases} m_1|m_2(\sigma)(X_i) & \text{if } m_1|m_2(\sigma)(X_i) \neq \sigma(X_i) \\ m_3(\sigma)X_i & \text{otherwise} \end{cases} \\ &= \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{else if } m_2(\sigma)(X_i) \neq \sigma(X_i) \\ m_3(\sigma)X_i & \text{otherwise} \end{cases} \end{aligned}$$

On the other hand, we have

$$\begin{aligned} m_1|(m_2)|m_3)(\sigma)(X_i) &= \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2|m_3(\sigma)X_i & \text{otherwise} \end{cases} \\ &= \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{else if } m_2(\sigma)(X_i) \neq \sigma(X_i) \\ m_3(\sigma)X_i & \text{otherwise} \end{cases} \end{aligned}$$

This proves that the sequential application of any two moves is associative. □

We can then define how a sequence of moves is applied on an assignment.

Definition 6. *Given a sequence of moves $M = [m_1, \ldots, m_k]$ and an assigment σ, we let the sequential application of M on σ be*

$$M(\sigma) = m_1|m_2|\ldots|m_k(\sigma)$$

4.1.2 Independence

When applying a move, we generally want the assignment to be modified as predicted. *Independent moves do not modify common variables.* It is thus natural to consider independent moves when selecting several moves.

Definition 7. *Two moves m_1 and m_2 are independent wrt σ iff $\forall X_i \in \mathcal{X} : m_1(\sigma)(X_i) \neq \sigma(X_i) \Rightarrow m_2(\sigma)(X_i) = \sigma(X_i)$. A set or sequence of moves M is independent wrt σ if every pairs of moves is independent.*

Independence among moves also implies commutativity, because the order of application of independent moves has no effect on the resulting assignment as it is showed in the following proposition.

Proposition 2. *If two moves m_1 and m_2 are independent, then $m_1|m_2(\sigma) = m_2|m_1(\sigma)$ for any assignments σ.*

Proof. We prove that if two moves m_1 and m_2 are independent, then $m_1|m_2(\sigma) = m_2|m_1(\sigma)$ for any assignments σ.

Let $\sigma \in \Lambda, X_i \in \mathcal{X}$,

$$m_1|m_2(\sigma)(X_i) = \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{otherwise} \end{cases}$$

$$= \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) = \sigma(X_i) \\ & \quad \text{and } m_2(\sigma)(X_i) \neq \sigma(X_i) \\ \sigma(X_i) & \text{otherwise} \end{cases}$$

By independence we have

$$m_2(\sigma)(X_i) \neq \sigma(X_i) \Rightarrow m_1(\sigma)(X_i) = \sigma(X_i)$$

we obtain

$$m_1|m_2(\sigma)(X_i) = \begin{cases} m_1(\sigma)(X_i) & \text{if } m_1(\sigma)(X_i) \neq \sigma(X_i) \\ m_2(\sigma)(X_i) & \text{if } m_2(\sigma)(X_i) \neq \sigma(X_i) \\ \sigma(X_i) & \text{otherwise} \end{cases}$$

By a similar reasoning, we obtain the same result for $m_2|m_1(\sigma)(X_i)$. This proves that for all variables X_i, $m_1|m_2(\sigma)(X_i) = m_2|m_1(\sigma)(X_i)$ for all assignments σ. And thus for all assignments σ,

$$m_1|m_2(\sigma) = m_2|m_1(\sigma)$$

□

The above proposition shows that given a sequence of independent moves $M = [m_1, \ldots, m_k]$, the moves $m_1, \ldots, m_k$ can be applied in any order. A sequence of independent moves can thus be represented by a set of independent moves. If M is a set of independent moves and σ is an assignment, we denote $M(\sigma)$ the assignment obtained by the successive application of the moves in M on σ. Thus a VLSN can be defined as follows

$VLSN(\mathcal{P}, \sigma) = \{M(\sigma) | M \subseteq \mathcal{M}$ is an independent set of moves wrt $\sigma\}$.

Example 4. *Consider the TSP (Section 2.1.1). The move $assign(X_i, j)$ assigns the value j to the variable X_i, so it only modifies the value assigned to the variable X_i. So this move is independent with another move $assign(X_k, m)$ if and only if $i \neq k$. If such moves are independent, the resulting assignment is independent from the order of the moves.*

Example 5. *Consider the GAP (Section 2.1.2). The exchange moves* $exchange(S_k, i, j)$ *and* $exchange(S_m, j, i)$ *are independent if and only if* $k \neq m$. *If so, when applying both of these moves, the order they are applied in is not important.*

For this same problem, suppose $8 \notin S_1$ *and consider the moves* $insert(S_1, 8)$ *and* $remove(S_1, 8)$. *These moves are not independent.*

4.1.3 Maintaining Partial Feasibility

We here address the first difficulty of selecting moves such that a structural constraint is not violated. We partition the constraints $\mathcal{C}$ of an COP into $\mathcal{C}_1 + \mathcal{C}_2$, where $\mathcal{C}_1$ is a global constraint capturing a core substructure of the COP and $\mathcal{C}_2$ are the remaining constraints. Typical examples of core constraints arising in VLSNs are permutation and partition constraints and moves are generally designed with these constraints in mind.

Move Graphs

Our approach considers more atomic moves than standard local search approaches. In standard local search approaches, only the moves respecting all the constraints $\mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2$ are considered in order to maintain the feasibility of the constraints. Here we deal with the feasibility of the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ differently. First we allow ourself to consider moves respecting $\mathcal{C}_2$, and not necessarily $\mathcal{C}_1$. So here we are more permissive than standard local search approaches. Then we select a subset of these moves such that the constraint $\mathcal{C}_1$ is respected.

MoveGraphs are key-components to search for a set of moves globally satisfying the constraints $\mathcal{C}_1$. Informally speaking, the edges in a MoveGraph represent moves such that a cycle represents a set of moves maintaining the feasibility of C_1 (even if a single move of such cycles may violate $\mathcal{C}_1$). The following definition of a MoveGraph is abstract. It will be instantiated later in this chapter for various global constraints and various neighborhoods by specifying what the nodes are and how the edges are mapped into moves.

Definition 8. *Given an assignment* σ, *a MoveGraph* $MG(\sigma)$ *is a labeled graph* $\langle V, E, \eta \rangle$ *where* η *is a function* $E \to \mathcal{M}$. *Given* $E' \subseteq E$, *we denote* $\eta(E') = \{\eta(i,j) | (i,j) \in E'\}$. *A move* $\eta(i,j)$ *is also denoted* η_{ij}.

The following definition captures the requirement that cycles maintain the feasibility of the global constraint C_1.

Definition 9. *Given an assignment σ, a MoveGraph $MG(\sigma)$ is cycle-consistent wrt the constraint $\mathcal{C}_1$ if*

$$\mathcal{C}_1(\eta(C)(\sigma)) = \mathcal{C}_1(\sigma)$$

for each cycle C in $MG(\sigma)$ such that $\eta(C)$ is independent wrt σ.

When considering cycle-consistent MoveGraphs, the neighborhood then becomes:

$$VLSN(\mathcal{P}, \sigma) = \\ \{\eta(C)(\sigma) | C \text{ is a cycle in } \ MG(\sigma) \wedge \eta(C) \text{ is independent wrt } \sigma\}.$$

Section 4.3 will present how to search this neighborhood by searching for cycles in MoveGraphs. Please note that the definition of MoveGraph is independent of any particular COP, so it is a useful concept in a constraint-based framework.

We now illustrate the concept of MoveGraph on two important global constraints in VLSN research: permutation and partitioning constraints.

Permutation Problems

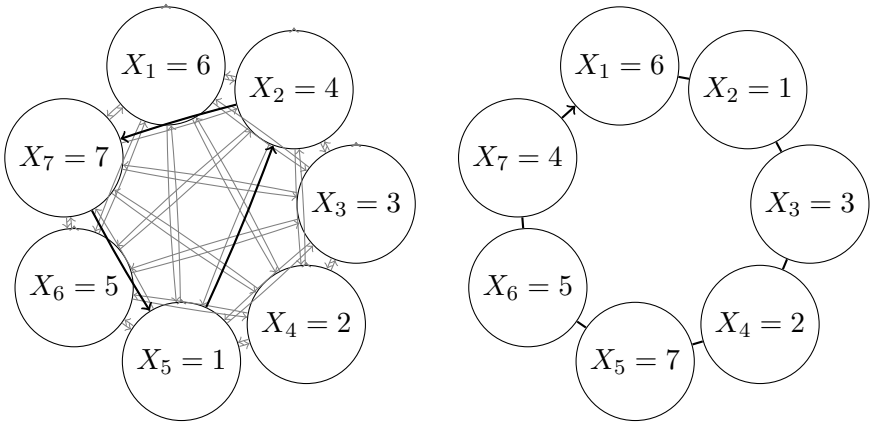
We now define a MoveGraph for permutation problems that considers moves $assign(X_j, i)$ (assign value i to variable X_j). Such moves break the permutation structure of an assignment if performed alone, but the MoveGraph will allow to select several of such moves such that the permutation structure is not broken after having applied all of them.

Definition 10. *Given a permutation problem on the variables $\mathcal{X} = [X_1, \ldots, X_k]$ and an assignment σ, the MoveGraph $MG_{perm}(\mathcal{X}, D, \sigma)$ is the label graph $\langle V, E, \eta \rangle$ where (1) $V = \mathcal{X}$, (2) $E = \{(X_i, X_j) : i \neq j\}$ and (3) $\eta(X_i, X_j) = assign(X_j, \sigma(X_i))$.*

In MG_{perm}, the nodes correspond to variables and the move associated with edge (X_i, X_j) assigns value $\sigma(X_i)$ to X_j. MG_{perm} is cycle-consistent with respect to the permutation constraint.

Proposition 3. *Given a permutation problem $\mathcal{P} = \langle f, \mathcal{C}_{perm} + \mathcal{C}_2, \mathcal{X}, D \rangle$, the MoveGraph $MG_{perm}(\mathcal{X}, D, \sigma)$ is cycle-consistent wrt $\mathcal{C}_{perm}$.*

Example 6. *Consider the TSP with $n = 7$ described in Example 1. Given the variables $\mathcal{X} = [X_1, \ldots, X_7]$ on domain $D = \{1, \ldots, 7\}$ and the assignment $\sigma = [6, 4, 3, 2, 1, 5, 7]$, the MoveGraph $MG_{perm}(\mathcal{X}, D, \sigma)$ is*



The cycle (X_5, X_2), (X_2, X_7), (X_7, X_5) *corresponds to the atomic moves* $assign(X_2, 1)$, $assign(X_7, 4)$ *and* $assign(X_5, 7)$. *These moves are independent and their application yields the new assignment* $\sigma' = [6, 1, 3, 2, 7, 5, 4]$. *The assignment* σ' *respects the permutation constraint although we applied moves that breaks this constraint if performed alone.*

Partitioning Problems

We now present a MoveGraph for partitioning problems such as the Generalized Assignment Problem. The MoveGraph considers three types of moves: $exchange(S_k, i, j)$ replaces the value j in variable S_k by value i, $insert(S_k, i)$ inserts the value i in S_k, and $remove(S_k, i)$ removes i from S_k. The nodes in the MoveGraph represent both variables and values, which enables us to encode the three types of moves.

Definition 11. *The MoveGraph* $MG_{part}(\mathcal{X}, D, \sigma)$ *for a partitioning problem and an assignment* σ *is the label graph* $\langle V, E, \eta \rangle$ *where*

(1) $V = \mathcal{X} \cup D$,

(2) $E = \{(i, j) \in V \times V | i \in D \vee j \in D\}$,

(3) (a) *For* $i, j \in D, \eta(i, j) = exchange(S_k, i, j)$ *with* $j \in \sigma(S_k)$,

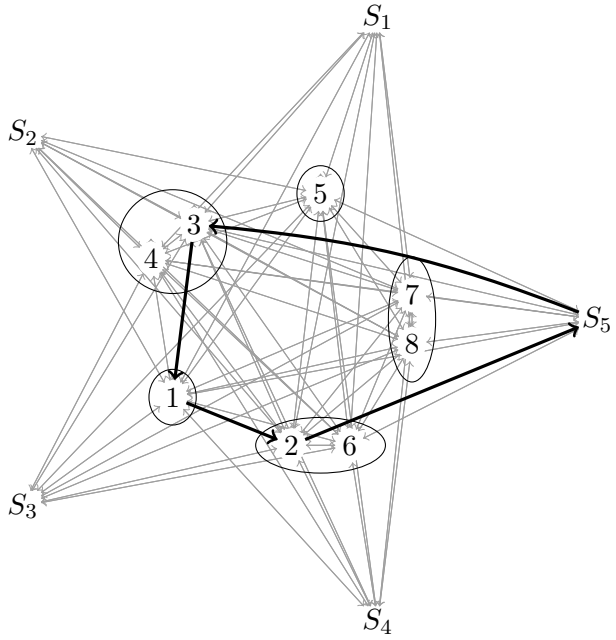
(b) *For* $i \in D$ *and* $S_k \in \mathcal{X}, \eta(i, S_k) = insert(S_k, i)$,

(c) *For* $i \in D$ *and* $S_k \in \mathcal{X}, \eta(S_k, i) = remove(S_l, i)$ *with* $i \in \sigma(S_l)$.

This MoveGraph is very similar to the graphs used in dedicated VLSN approaches for partitioning problems.
This MoveGraph is cycle-consistent with respect to the partitioning constraint. Please note that the semantic of the move represented by an edge (S_k, i) does not depend on S_k.

Proposition 4. *Given a partitioning problem* $\langle f, \mathcal{C}_{part} + \mathcal{C}_2, \mathcal{X}, 2^D \rangle$ *and an assignment* σ*, the MoveGraph* $MG_{part}(\mathcal{X}, D, \sigma)$ *is cycle-consistent wrt* $\mathcal{C}_{part}$*.*

Example 7. *Consider the GAP introduced in Example 2 with* $n = 8$ *and* $K = 5$*. For the assignment* $\sigma = \{S_1 = \{5\}, S_2 = \{3, 4\}, S_3 = \{1\}, S_4 = \{2, 6\}, S_5 = \{7, 8\}\}$*, the MoveGraph* $MG_{part}(\mathcal{X}, D, \sigma)$ *is*



The cycle $(1, 2), (2, S_5), (S_5, 3), (3, 1)$ *corresponds to the following moves:* $exchange(S_4, 1, 2)$*,* $insert(S_5, 2)$*,* $remove(S_2, 3)$ *and* $exchange(S_3, 3, 1)$*. Notice that the move* $remove(S_2, 3)$ *labels all the arcs* $\{(S_k, 3) : \forall k = 1, \ldots, K\}$*. These four moves are independent and their application yields the new assignment* $\sigma' = \{S1 = \{5\}, S2 = \{4\}, S3 = \{3\}, S4 = \{1, 6\}, S5 = \{2, 7, 8\}\}$*. Notice that the assignment* σ' *still respects the partition constraint although each of the single applied moves violates it.*

4.1.4 Ensuring Efficient Search of the Cyclic Neighborhood

Compositionality

Computing the differentiation of a set of moves on the constraints and the objective is complex in general, as it may require simulation. We now define the concept of compositional moves. *When only sets of compositional moves are considered, very good candidates in the VLSN can*

be searched efficiently. Informally speaking, moves are compositional if the differentiation of a set of moves is the sum of the differentiation of each individual move. In the following definition, for a set M of independent moves, we use $\Delta_f(M,\sigma)$ to denote $f(M(\sigma)) - f(\sigma)$ and $\Delta_{\mathcal{C}_2}(M,\sigma)$ to denote $\mathcal{C}_2(M(\sigma)) - \mathcal{C}_2(\sigma)$.

Definition 12. *Given an COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, a set of independent moves M is compositional wrt a solution σ if*

(1) $\Delta_{\mathcal{C}_2}(M,\sigma) = \sum_{m\in M} \Delta_{\mathcal{C}_2}(m,\sigma)$

(2) $\Delta_f\ (M,\sigma) = \sum_{m\in M} \Delta_f\ \ (m,\sigma)$

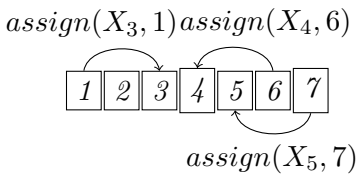
It is easy to compute the impact of a set of compositional moves on the constraints or on the objective. This allows the design of polynomial-time heuristics for searching the following neighborhood,

$$VLSN^A(\mathcal{P},\sigma) = \{\eta(C)(\sigma) \mid C \text{ is a cycle in } MG(\sigma) \text{ and } \eta(C) \text{ is independent wrt } \sigma \text{ and } \eta(C) \text{ is compositional wrt } \sigma\}.$$

Note that the size of the neighborhood is still exponential. Moreover, if $LS(\mathcal{P},\sigma)$ denotes the neighborhood used in standard local search approaches (selecting only one move), we still have

$$LS(\mathcal{P},\sigma) \subseteq VLSN^1(\mathcal{P},\sigma)$$

Example 8. *Consider the TSP with $n = 7$ and illustrated below. Let the initial solution be $\sigma_0 = [1,2,3,4,5,6,7]$ with the cost $c_{12}+c_{23}+c_{34}+c_{45}+c_{56}+c_{67}+c_{71}$.*



Consider the moves $m_1 = assign(X_3, 1)$, $m_2 = assign(X_4, 6)$ and $m_3 = assign(X_5, 7)$. The following array gives the cost of the assignment resulting from the application of some combinations of these moves.

M	$f_{TSP}(M(\sigma_0))$	$\Delta_{f_{TSP}}(M, \sigma_0)$
$\emptyset$	$c_{12} + c_{23} + c_{34} + c_{45} + c_{56} + c_{67} + c_{71}$	0
$\{m_1\}$	$c_{12} + c_{21} + c_{14} + c_{45} + c_{56} + c_{67} + c_{71}$	$-c_{23} - c_{34} + c_{21} + c_{14}$
$\{m_2\}$	$c_{12} + c_{23} + c_{36} + c_{65} + c_{56} + c_{67} + c_{71}$	$-c_{34} - c_{45} + c_{36} + c_{65}$
$\{m_3\}$	$c_{12} + c_{23} + c_{34} + c_{47} + c_{76} + c_{67} + c_{71}$	$-c_{45} - c_{56} + c_{47} + c_{76}$
$\{m_1, m_2\}$	$c_{12} + c_{21} + c_{16} + c_{65} + c_{56} + c_{67} + c_{61}$	$-c_{23} - c_{34} - c_{45} + c_{21} + c_{16} + c_{65}$
$\{m_1, m_3\}$	$c_{12} + c_{21} + c_{14} + c_{47} + c_{76} + c_{67} + c_{61}$	$-c_{23} - c_{34} - c_{45} - c_{56} + c_{21} + c_{14} + c_{47} + c_{76}$

The third column of this array shows that the moves m_1 and m_2 are not compositional, because

$$\Delta_{f_{TSP}}(\{m_1, m_2\}, \sigma_0) \neq \Delta_{f_{TSP}}(m_1, \sigma_0) + \Delta_{f_{TSP}}(m_2, \sigma_0)$$

However the moves m_1 and m_3 are compositional because

$$\Delta_{f_{TSP}}(\{m_1, m_3\}, \sigma_0) = \Delta_{f_{TSP}}(m_1, \sigma_0) + \Delta_{f_{TSP}}(m_3, \sigma_0)$$

Improvement Graph

The neighborhood $VLSN^A(\mathcal{P}, \sigma)$ can now be searched efficiently through the concept of Improvement Graph, that is built automatically from the MoveGraph. *Improvement graphs are the core of VLSN algorithms but in constraint-based VLSN, they can be derived automatically from MoveGraphs thanks to the differentiability of the moves.* The key idea is to (1) remove edges (i, j) with $\Delta_{\mathcal{C}_2}(\eta_{ij}, \sigma) \neq 0$ as these moves violate constraint $\mathcal{C}_2$ and (2) add a weight $\Delta_f(\eta_{ij}, \sigma)$ on every edge (i, j).

Definition 13. *Given an COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, a MoveGraph $G = \langle V, E, \eta\rangle$ and an assignment σ, the improvement graph is the weighted graph $IG(G, \sigma) = (V, E', \eta, w)$ such that*

(1) $E' = \{(i, j) \in E | \Delta_{\mathcal{C}_2}(\eta_{ij}, \sigma) = 0\}$,

(2) $w_{ij} = \Delta_f(\eta_{ij}, \sigma)$.

The neighborhood $VLSN^A(\mathcal{P}, \sigma)$ can then be searched more efficiently by searching for cycles in the improvement graph, for the two following reasons. First, pruning the set of edges does not restrict the neighborhood. Indeed the hard constraint are satisfied by all solutions: $\mathcal{C}_2(\sigma) = 0$, thus $\Delta_{\mathcal{C}_2}(m, \sigma) \geq 0, \forall m \in \mathcal{M}$. So $\sum_{m \in M} \Delta_{\mathcal{C}_2}(m, \sigma) = 0 \iff$

$\Delta_{\mathcal{C}_2}(m, \sigma) = 0, \forall m \in M$. Thus any compositional cycle C in the MoveGraph G such that $\Delta_{\mathcal{C}_2}(\eta(C), \sigma) = 0$ will also be in the corresponding improvement graph.

Second, precomputing the values $\Delta_f(m, \sigma)$ allows to compute the variations of a set of compositional moves represented by a path in the improvement graph very efficiently from Definition 12.

Note that in order to compute the improvement graph, we only need to compute $\Delta_{\mathcal{C}_2}(m, \sigma)$ and $\Delta_f(m, \sigma)$ for all moves m considered. This also has to be done in a standard local search algorithm. Thus the time-complexity of building the improvement graph is the same as searching standard local search neighborhoods.

Incremental Update of the Improvement Graph

Because the improvement graph depends on the current solution σ, it must be updated at each iteration. VLSN algorithms update the improvement graph incrementally, however the set of edges to update is problem-dependent. *Fortunately, in CBVLSN, the set of edges to update can be derived automatically.* Indeed, one needs only to consider the edges that are not compositional with the moves applied at the previous iteration.

Proposition 5. *Given an COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, a MoveGraph $G = \langle V, E, \eta\rangle$ and an assignment σ, let M be a set of compositional moves wrt σ and η_{ij} be a move compositional with all moves in M wrt σ. We have*

$$\Delta_f(\eta_{ij}, M(\sigma)) = \Delta_f(\eta_{ij}, \sigma) \tag{4.1}$$
$$\Delta_{\mathcal{C}_2}(\eta_{ij}, M(\sigma)) = \Delta_{\mathcal{C}_2}(\eta_{ij}, \sigma) \tag{4.2}$$

Proof.

$$\begin{aligned}
\Delta_f(\eta_{ij}, M(\sigma)) &= f(\eta_{ij}(M(\sigma))) - f(M(\sigma)) \\
&= \left(f(\sigma) + \Delta_f(\eta_{ij}, \sigma) + \sum_{m \in M} \Delta_f(m, \sigma) \right) \\
&\quad - \left(f(\sigma) + \sum_{m \in M} \Delta(m, \sigma) \right) \\
&= \Delta_f(\eta_{ij}, \sigma)
\end{aligned}$$

The same reasoning holds with $\mathcal{C}_2$. □

Proposition 5 shows that the presence and the cost of an edge (i, j) in the improvement graph is constant if applying only compositional moves wrt η_{ij}. After having applied a set M of moves, an edge (i, j) has to be updated only if η_{ij} is not compositional with M, or if the semantic of η_{ij} has changed.

Let us review what we have achieved so far. We have shown that VLSNs can be formalized abstractly in terms of differentiable constraints and functions, and a partitioning of the constraints into a global constraint capturing the important substructure of the problem and other, possible global, constraints. To ensure feasibility, we introduced the concept of cyclic-consistent MoveGraph, which guarantees that cycles in the MoveGraph maintain the feasibility of the distinguished global constraint. Finally, we have indicated that the differentiation on the model of a set of moves, if it is restricted to be compositional, can be computed very easily.

The only remaining issues are how to search the cyclic neighborhood and how to test independence and compositionality. One possibility is to implement directly Definition 12. Such a "simulation" approach is often orders of magnitude slower than a dedicated implementation. Our approach however computes compositionality incrementally from a small extension in the CLBS interface of constraints and functions.

4.2 Automatic Derivation of Independent and Compositional Moves

This section presents how to derive systematically the independence and compositionality of moves from a small extension in the CBLS interface. *Input and output variables are two fundamental concepts used to derive a sufficient condition for a set of moves to be compositional and independent.* The approach relies mainly on two observations: (1) the independence and compositionality of two moves only depends on the current values of the decision variables; and (2) the differentiation typically depends on the values of a subset of the decision variables.

4.2.1 Definitions

The first concept, i.e., output variables, allows us to determine whether two moves are independent.

Definition 14. *Given a set of variables $\mathcal{X}$ and an assignment σ, the output variables of a move m, denoted* $\mathrm{Var}^{\neq}(m, \sigma)$*, is the set of the*

variables modified by applying the move m on σ: $\mathrm{Var}^{\neq}(m,\sigma) = \{X_i \in \mathcal{X} : m(\sigma)(X_i) \neq \sigma(X_i)\}$.

Definition 15. *Given two moves m_1 and m_2 and an assignment σ, m_1 and m_2 are variable-independent wrt σ iff* $\mathrm{Var}^{\neq}(m_1,\sigma) \cap \mathrm{Var}^{\neq}(m_2,\sigma) = \emptyset$.

If two moves are variable-independent wrt σ, they are also independent wrt σ, and vice-versa.

Proposition 6. *Given two moves m_1 and m_2 and an assignment σ, m_1 and m_2 are variable-independent wrt σ iff m_1 and m_2 are independent wrt σ.*

The following definition is necessary to define variable-compositionality.

Definition 16. *Given a set of variables $\mathcal{X}$ and a subset of these variables $X \subseteq \mathcal{X}$, two assignments σ_1, σ_2 are X-equivalent if $\forall X_i \in X : \sigma_1(X_i) = \sigma_2(X_i)$.*

Now, testing compositionality requires to determine which variables would change the differentiation of moves. This intuition is captured by the concept of input variables.

Definition 17. *Given a set of variables $\mathcal{X}$, a moves m, an assignment σ and a function $g : \Lambda \to \mathbb{Z}$, the input variables $Var^{<}(g,m,\sigma)$ is a subset of $\mathcal{X}$ of minimum cardinality such that for all assignments $\sigma' \in \Lambda$ such that σ and σ' are $Var^{<}(g,m,\sigma)$-equivalent we have*

$$\Delta_g(m,\sigma') = \Delta_g(m,\sigma)$$

Example 9. *Consider the variables $\mathcal{X} = [X_1, X_2, X_3]$, the domain $D = \{0,1\}$ and the function $f(\sigma) = \sigma(X_1).\sigma(X_2).\sigma(X_3)$. Let the assignment $\sigma = \{X_1 = X_2 = X_3 = 0\}$. Clearly $\Delta_f(assign(X_1,1),\sigma) = 0$. This holds as long as $X_2 = 0$ or $X_3 = 0$. Thus the input variables $\mathrm{Var}^{<}(assign(X_1,1),\sigma)$ can be either $\{X_2\}$ or $\{X_3\}$. This shows that the input variables may not be unique. Other definitions of input variables may lead to unicity, but for reasons that are made clear hereafter, we are interested in having as few input variables as possible.*

Definition 18. *Given a function g and an assignment σ, a set of moves $M = \{m_1, \ldots, m_k\}$ is variable-compositional wrt g and σ iff*

$$Var^{<}(g,m_i,\sigma) \cap \mathrm{Var}^{\neq}(m_j,\sigma) = \emptyset \qquad \forall i,j \in \{1,\ldots,k\} \text{ with } i \neq j$$

We can now define a sufficient condition for a set of moves to be compositional.

Proposition 7. *Given an COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, an assignment σ and a set of independent moves M, if each pair of moves in M is variable-compositional wrt f and σ, and variable-compositional wrt $\mathcal{C}_2$ and σ, then M is compositional wrt σ.*

Proof. Let $M = \{m_1, \ldots, m_k\}$ be a set of moves variable-compositional wrt f and σ, and variable-compositional wrt $\mathcal{C}_2$ and σ. We let $M_j = \{m_1, \ldots, m_j\}$ and $\sigma_j = M_j(\sigma)$ for all $j = 1, \ldots, k$. We have to prove that

(A) $\Delta_f(M, \sigma) = \sum_{i=1}^{k} \Delta_f(m_i, \sigma)$

(B) $\Delta_{\mathcal{C}_2}(M, \sigma) = \sum_{i=1}^{k} \Delta_{\mathcal{C}_2}(m_i, \sigma)$

We prove (A). A similar reasoning proves (B). From Definition 18 we have

$$\mathrm{Var}^{<}(f, m_j, \sigma) \cap \mathrm{Var}^{\neq}(m_i, \sigma) = \emptyset \qquad \forall i, j \in \{1, \ldots, k\} \text{ with } i \neq j \tag{4.3}$$

So for all $j = 2, \ldots, k$, we have $\sigma_{j-1}(X_l) = \sigma(X_l)$ for all $X_l \in \mathrm{Var}^{<}(f, m_j, \sigma)$, and thus σ_{j-1} and σ are $\mathrm{Var}^{<}(f, m_j, \sigma)$-equivalent assignments. From Definition 17 we obtain

$$\Delta_f(m_j, \sigma_{j-1}) = \Delta_f(m_j, \sigma) \tag{4.4}$$

Moreover we have

$$\begin{aligned}
\Delta_f(M_j, \sigma) &= f(M_j(\sigma)) - f(\sigma) && (4.5)\\
&= f(m_j(\sigma_{j-1})) - f(\sigma) && (4.6)\\
&= f(m_j(\sigma_{j-1})) - f(\sigma_{j-1}) + f(\sigma_{j-1}) - f(\sigma) && (4.7)\\
&= \Delta_f(m_j, \sigma_{j-1}) + f(M_{j-1}(\sigma)) - f(\sigma) && (4.8)\\
&= \Delta_f(m_j, \sigma_{j-1}) + \Delta_f(M_{j-1}, \sigma) && (4.9)
\end{aligned}$$

Thus, by (4.4), we obtain

$$\Delta_f(M_j, \sigma) = \Delta_f(M_{j-1}, \sigma) + \Delta_f(m_j, \sigma) \qquad \forall j = 2, \ldots, k \tag{4.10}$$

This recurrence formula leads to

$$\Delta_f(M_j, \sigma) = \sum_{i=1}^{j} \Delta_f(m_i, \sigma) \qquad \forall j = 1, \ldots, k \tag{4.11}$$

Finally we have $\Delta_f(M, \sigma) = \Delta_f(M_k, \sigma) = \sum_{i=1}^{k} \Delta_f(m_i, \sigma)$. □

Input and output variables are two fundamental concepts: given a COP, they allow us to compute whether a set of moves is surely compositional and independent, without any additional knowledge from the user. This enables the implementation of arbitrarily complex VLSNs with any constraint that can be searched by generic algorithms. It suffices to extend the CBLS interface for constraints and functions to include, not only violations and differentiation, but also input and output variables which is natural in practice. Then it is natural to approximate the compositional and independent cyclic neighborhood by the following neighborhood,

$$VLSN^B(\mathcal{P},\sigma) = \{\eta(C)(\sigma) \mid C \text{ is a cycle in } MG(\sigma) \text{ and } \eta(C) \text{ is variable-independent and variables-compositional wrt } \sigma, f \text{ and } \mathcal{C}_2\}.$$

How to search $VLSN^B(\mathcal{P},\sigma)$ is described in Section 4.3.

Example 10. *We illustrate here all the concepts presented in this section on the GAP. We will illustrate the input variables by considering the capacity constraint that ensures that* $\sum_{i\in\sigma(S_k)} b_k \leq B, \forall k = 1,...,K$. *We consider the move* $insert(S_k, i)$ *that inserts the element* i *in the set variable* S_k *(*$i \notin S_k$*). The output variables of such move is thus* $\text{Var}^{\neq}(insert(S_k,i),\sigma) = \{S_k\}$. *In order to check whether this move respects the capacity constraint, we must check that* $\sum_{j\in\sigma(S_k)} b_j + b_i \leq B$. *Indeed, the other variables remaining unchanged, there is no need to verify the capacity constraint for all other sets. In order to check whether the insert move respects the capacity constraint, only the current value of the variable* S_k *has to be considered. So* $\text{Var}^{<}(insert(S_k), i) = \{S_k\}$. *This tells* whether the move $insert(S_k, i)$ respects the capacity constraint or not is independent from the current value of all other variables than S_k.

Now if the current solution σ *is such that* $\sum_{i\in\sigma(S_k)} b_i = 10$ *and* $B = 11, b_i = 1, b_j = 1$ *with* $i, j \notin S_k$, *then both moves* $insert(S_k, i)$ *and* $insert(S_k, j)$ *respect the capacity constraint separatedly. However, if performed together, the capacity constraint will be violated. Because some of the output variables of one move is an input variable of the other move, our framework knows there is a risk that the constraint may be violated despite the fact that it is respected by both moves separatedly. So our generic search algorithm won't select such pair of moves together.*

4.2.2 Operational advantages

Constraint and objective operators: Once the input and output variables are available for basic constraints and objectives, the input variables can also be synthesized for traditional logical and arithmetic operators.

Proposition 8. *The input variables can be determined for the sum, product or other operations α between multiple functions:*

$$Var^{<}(f \,\alpha\, g, m, \sigma) \subseteq Var^{<}(f, m, \sigma) \cup Var^{<}(g, m, \sigma)$$

As a result, search algorithms can be enhanced systematically to compute independence and compositionality on the fly: They only need to collect the input and output variables of the moves for the different differential invariant used in the model, and then apply Propositions 7 and 8.

Checking variable-compositionality and variable-independence: Checking variable-compositionality and variable-independence is a crucial step in CBVLSN search algorithms. We here describe how this check can be efficiently computed. Let a set M of moves and an assignment σ, checking whether a move m is variable-compositional and variable-independent with M can be done in $\mathcal{O}(|M|.o_V + o_V)$, where o_V is an upper bound of the number of input and output variables per move.

This check is performed in two steps. First the input and output variables of the moves in M are marked. Introduce two boolean arrays $inputMarked$ and $outputMarked$, both indexed by the variables in $\mathcal{X}$. Marking a move m means setting to true the cells of these arrays corresponding to $\text{Var}^{<}(m, \sigma)$ and $\text{Var}^{\neq}(m, \sigma)$. This can be done in $\mathcal{O}(o_V)$ where o_V is an upper bound of the number of input and output variables per move. Marking all the nodes in M can thus be done in $\mathcal{O}(|M|.o_V)$.

Second, once the moves are marked, it is easy to check whether a move m is variable-independent and variable-compositional with M. It suffices to check whether no output variable in $\text{Var}^{\neq}(m, \sigma)$ is marked in both arrays, and if no input variables is marked in $outputMarked$. This check can be done in $\mathcal{O}(o_V)$.

Update of the improvement graph: Once a set M of moves is selected and applied, the improvement graph must be recomputed. Section 4.1.4 showed that only moves non-compositional with M have to be reconsidered during this update. As Proposition 7 stated, variable-compositionality is stricter than compositionality. We can thus update an improvement graph (V, E', η, w) by only considering non-variable-compositional moves wrt M; only the edges in the set $conflict(M, \sigma) = \bigcup_{m \in M} \{(i,j) \in E : \text{Var}^{<}(\eta_{ij}, \sigma) \cap \text{Var}^{\neq}(m, \sigma) \neq \emptyset\}$ have to be reconsidered.

In summary, we have presented a theory for constraint-based VLSN. Given a CBLS interface for constraints and objective functions enhanced

with the concepts of input and output variables for each move, we have demonstrated that a generic VLSN implementation can be derived systematically from a cycle-consistent MoveGraph associated with $\langle f, \mathcal{C}_1+\mathcal{C}_2, \mathcal{X}, D\rangle$. A constraint-based VLSN framework will then provide a library of MoveGraphs and differential invariants that can be extented by the users by implementing their own. These contributions make it possible to define VLSN models at a high level of abstraction and to include idiosyncratic constraints naturally.

Node MoveGraphs A Node MoveGraph is a MoveGraph with the input and output variables assigned to the nodes, rather than on the edges. Indeed each move has its own input and output variables. From the point of vue of the Move Graph, the input and output variables can thus be assigned to the edges. However in many applications, the input and output variables can also be assigned on the node of the MoveGraph. This arises if all edges incident to a particular node represent moves having the same input and output variables. This the case of the GAP for example. In such cases, we define

- $\text{Var}^{<}{}_V(f, i, \sigma)$ as the input variables assigned to node $i \in V$ wrt the function f and assignment σ
- $\text{Var}^{\neq}{}_V(i, \sigma)$ as the output variables assigned to node $i \in V$ wrt the assignment σ.

Definition 19. *A Node MoveGraph* $NMG(\sigma) = \langle V, E, \eta, \text{Var}^{<}{}_V, \text{Var}^{\neq}{}_V\rangle$ *wrt the COP* $\langle f, \mathcal{C}_1+\mathcal{C}_2, \mathcal{X}, D\rangle$ *is a MoveGraph* $\langle V, E, \eta\rangle$ *such that* $\forall (i,j) \in E$

1. $\text{Var}^{<}(f, \eta_{ij}, \sigma) \subseteq \text{Var}^{<}{}_V(f, i, \sigma) \cup \text{Var}^{<}{}_V(f, j, \sigma)$
2. $\text{Var}^{<}(\mathcal{C}_2, \eta_{ij}, \sigma) \subseteq \text{Var}^{<}{}_V(\mathcal{C}_2, i, \sigma) \cup \text{Var}^{<}{}_V(\mathcal{C}_2, j, \sigma)$
3. $\text{Var}^{\neq}(\eta_{ij}, \sigma) \subseteq \text{Var}^{\neq}{}_V(i, \sigma) \cup \text{Var}^{<}{}_V(j, \sigma)$

The Node MoveGraphs are especially useful for implementation optimization. Theoretically, any Node MoveGraph can be represented as a MoveGraph.

4.3 Exploring the Cyclic Neighborhood

This section describes how the cyclic neighborhood $VLSN^B(\mathcal{P}, \sigma)$ can be searched efficiently despite its theoretical complexity.

Proposition 9. *Finding the best candidate in $VLSN^B(\mathcal{P}, \sigma)$ is NP-Hard.*

Proof. This is proved by observing that in the particular case of the Generalized Assignment Problem, finding the best candidate in $VLSN^B(\mathcal{P}, \sigma)$ is reduced to solving the Cycle Through Distinct Subpartition Problem (CTDSP), that has been proven to be NP-Hard [TO89].. The general problem of searching this neighborhood is thus NP-Hard. Notice that the CTDSP is the usual subproblem to be solved in cyclic VLSN dedicated to partitioning problems.

□

The complexity of searching the best candidate in $VLSN^B(\mathcal{P}, \sigma)$ contrasts with the polynomial-time complexity of searching the minimum-cost cycle in a graph. However an efficient heuristic can be designed, based on a polynomial-time algorithm searching for cycles. A heuristic computing solutions of the Cycle Through Distinct Subpartition problem has been presented in [AOS01], based on a label-correcting algorithm. Hereafter we present a variation of this heuristic to find solutions of the more general problem of searching $VLSN^B(\mathcal{P}, \sigma)$. This heuristic is depicted in Algorithm 5.

This heuristic is based on the label-correcting algorithm [AMO93] solving the shortest-path problem. This label-correcting algorithm can find shortest paths in a graph with edges with negative cost. This algorithm computes the shortest-paths from a start node s to any other node in the graph. For all nodes i, the algorithm maintains the predecessor $P[i]$ of node i on the shortest-path from s to i found so far. It also maintains $d[i]$, the distance from s to i according to the paths encoded by the predecessors. If $d[i] + w_{ij} \geq d[j]$ for all edges $(i, j) \in E$, it can be proved that the paths are optimal. To ensure this condition, the algorithm maintains a LIST that contains all the nodes i such that an outgoing edge may violate the optimality condition. Once LIST is empty, the algorithm stops. Until then, the algorithm pops a node from LIST, examine all outgoing edges (i, j) and sets the predecessors of j as i if $d[i] + w_{ij} < d[j]$.

Cycles may be identified with the label-correcting algorithm by walking back the path from s to i when popping the node i from LIST. There is a cycle in the shortest-path tree encoded by the predecessors if we reach a node twice during this walk.

The label-correcting algorithm is extended to only consider independent and compositional cycles. The first check is done when the

```
1  d(s) := 0; pred(s) := 0; d(j) := ∞, ∀j ≠ s;
2  LIST = {(s, 0)};
3  while LIST ≠ ∅ do
5      remove an element (k, i) in LIST;
6      mark the nodes and edges in P[i];
8      if P[i] is not independent or not compositional, then
       continue;
9      for (i, j) ∈ E do
11         if j ∈ P[i] then return the subpath from i to j in P[i] ;
13         else if η_ij is independent and compositional with all
           moves in P[i] then
15             d(j) := d(i) + w_ij;
16             pred(j) := i;
18             if (j, k) and (j, k + 1) ∉ LIST then add (k + 1, j) in
               LIST ;
```

Algorithm 5: Heuristic algorithm searching for independent and compositional cycles in an Improvement Graph (i.e. a MoveGraph with precomputed values). The label-correcting algorithm for searching for paths [AMO93] is represented in non-bold font. The added lines are in bold. These lines allows to return a cycle only if it represents a set of independent and compositional moves.

algorithm found an edge such that $d[i] + w_{ij} < d[j]$. Such edge should be used to update the shortest-path to j. However the algorithm also check whether this edge is independent and compositional with all edges in the shortest-path from s to i. If not, the shortest-path to j is not updated.

The second check is done when a node i is popped from LIST. The algorithm checks whether the shortest-path to i is still independent and compositional. Indeed, when the shortest-path to a node j is updated, the shortest-paths to all successors of j are also modified, and some of these paths may not be independent and compositional.

The complexity of Algorithm 5 depends on the implementation of `LIST`. If we use a queue (FIFO), then Algorithm 5 achieves a polynomial-time complexity.

Proposition 10. *Given an COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, X, D\rangle$, an assignment σ and an improvement graph $G = (V, E, \eta, w)$, let $n = |V|$, $m = |E|$, o_V an upper bound on the number of input and output variables per move, and U be the maximal cardinality of any path in the shortest path tree. The time-complexity of Algorithm 5 is $\mathcal{O}(nU^2 o_V + mU(U + o_V))$ if using a FIFO implementation for* `LIST`.

Proof. First note that when popping a couple (k, i) from `LIST`, only couples of the form $(k + 1, j)$ will be added to `LIST`. As we use a FIFO, couples (k, i) will thus be considered in increasing value of k.

After popping (k, i), only couples (k', i') with $k' > k$ will be added to `LIST`. So once (k, i) is popped, it won't be added to `LIST` again, and thus cannot be popped from `LIST` twice. Given a couple $(k, i) \in LIST$, k represents the cardinality of the shortest path from s to i when (k, i) was added into `LIST`. So U is an upper bound of the value of k. The while loop is thus executed at most nU times.

Inside the while loop, marking the nodes and edges in $P[i]$ can be done in $\mathcal{O}(U.o_V)$ (Section 4.2 describes how). Checking independence and compositionality of the corresponding moves is done during the marking. So line 8 takes $\mathcal{O}(U.o_V)$. Thus the complexity of lines 5-8 is $\mathcal{O}(nU^2 o_V)$. Each edge (i, j) can be considered only when a couple (k, i) is popped. Thus each edge can be considered at most U times. Line 11 takes $\mathcal{O}(U)$, checking whether η_{ij} can be added to $P[i]$ in $\mathcal{O}(o_V)$ and all operations in lines 15-18 take constant time. Thus the total complexity of the lines inside the for loop is $\mathcal{O}(mU(U + o_V))$.

These considerations lead to a complexity of $\mathcal{O}(nU^2 o_V + mU(U + o_V))$ for Algorithm 5.

□

Example 11. *This complexity is considerably lowered for practical problems. For the GAP, each move modifies one variable, and this variable is also the unique variable in the input variables. This has two consequences. First no more than K moves can be independent (as each move modifies one variable). This leads to $U = K$. Second, $o_V = \mathcal{O}(1)$. So the complexity of our algorithm 5 is $\mathcal{O}(nK^2 + mK^2)$. Note that if we stop at the first cycle found, then the complexity becomes $\mathcal{O}(nK^2+mK)$. So our algorithm has the same theoretical complexity as the algorithm presented in [AOS01]. Indeed, for this problem, our algorithm performs the same operations as in [AOS01].*

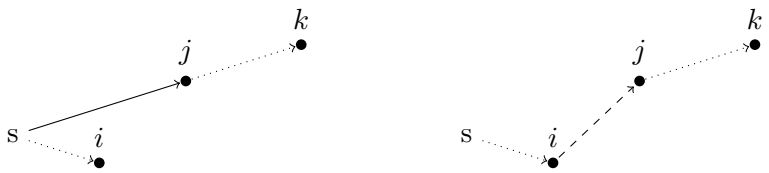


Figure 4.1: Illustrating the Behavior of the Algorithm.

Figure 4.1 illustrates why the algorithm checks for independence and compositionality twice and why the algorithm is incomplete. Consider Algorithm 5 when a node i has been popped and the edge (i, j) is considered to be added in the shortest path tree (line 7). In this example, all paths in this tree were independent and compositional, although the moves represented by the edges (s, i) and (j, k) are not. The edge (i, j) is added to the shortest path tree because the path $[(s, i), (i, j)]$ is independent and compositional (line 9). This implicitely change the shortest path from s to k to be $[(s, i), (i, j), (j, k)]$, that is not independent and compositional. This illustrates why the shortest path tree may contain paths that are not independent and compositional and why we need to check for it when we pop a node from the list. After the addition of the edge (i, j), the edge (s, j) is removed from the shortest path tree. The path $[(s, j), (j, k)]$ is thus forgotten while the current path from s to k is not valid (not independent and compositional). This illustrates why Algorithm 5 is incomplete, since it has lost trak of the path from s to k going through j.

Chapter 5

Implementation

We implemented generic abstractions to efficiently compute the differentiation of several moves, based on the concept of input and output variables. Then, using these abstractions, we implemented generic VLSN search procedures selecting several moves at each iteration, such that a structural global constraint is respected. These VLSN search procedures are concise: they can be very quickly implemented and modified to take into account some expert's knowledge of the problem.

In this chapter, first we detail how we extended the CBLS solver of Comet. Second we explain how the input and output variables, and the MoveGraphs are described. Then we present our abstractions checking compositionality and independence and differentiating a set of moves. Finally we describe our generic VLSN search procedures.

5.1 An extension of the CBLS solver

We introduce two new concepts to the current Constraint-Based Local Search solver of Comet(Listing 5.1). First the concept of decision variables, which are variables not maintained by invariants [HM06]. Decision variables are important for memory efficiency.

Second the concept of iteration, which is the step of selecting several moves and to apply them. This concept is a key component of several data-structures that can be incrementally updated.

5.1.1 Introducing Decision Variables

A Decision variable is a variable whose value is not a function of some others. Such variables are thus necessary to fully describe a solution. In Comet, invariants efficiently maintain the value of some variables

depending on some others. In terms of implementation, we thus define a decision variable as a variable that is not maintained by mean of an invariant. In the CBLS module of COMET, each variable has an unique LS identifier. One of the role of our VLSN solver is to assign a second VLSN identifier to each decision variable such that all decision variables have consecutive VLSN identifiers.

The set of LS identifiers of all the decision variables is stored in the variable `decVariables` (Listing 5.1). The range `decVars` contains the consecutive VLSN identifiers of the decision variables, and the array `idsa` contains the VLSN identifier of decision variables: `idsa[lsID]` represents the VLSN identifiers of the variable with the LS identifier `lsID` (`-1` if the variable is not a decision variable).

These values are set in the two following methods. The first method `registerDecisionVariable` is used to indicate to the solver that a variable is a decision variable. Second, `closeVariables` is called when all the decision variables have been registered and it computes the array `idsa`.

5.1.2 Introducing Iterations

The concept of iteration is used to incrementaly maintain the differentiation of atomic moves wrt changes in the solution. The Solver has a counter `iter` (Listing 5.1) indicating the current iteration. This counter is incremented when a move is performed (method `performMove`). The array `lastModified` is indexed by VLSN identifiers of the decision variables and indicates the last iteration at which the decision variables were modified. We use events [HM05b] to change the values of this array when the decision variables are modified (methods `registerDecisionVariable`).

```
1  class Solver<VLSN> extends Solver<LS>{

     set{int} decVariables;
     range decVars;
     int[] idsa;
6
     Integer iter;
     int[] lastModified;

     Solver<VLSN>() : Solver<LS>(){
11     decVariables = new set{int}();
       iter = new Integer(0);
     }
     void registerDecisionVariable(var{int} v){
       int c = decVariables.getSize();
```

```
16      decVariables.insert(v.getId());
        if ( c < decVariables.getSize())
          whenever v@changes(){ setModified(v.getId()); }
      }
      void registerDecisionVariable(var{set{int}} v){
21      int c = decVariables.getSize();
        decVariables.insert(v.getId());
        if ( c < decVariables.getSize()){
          whenever v@insert(int i){setModified(v.getId()); }
          whenever v@remove(int i){setModified(v.getId()); }
26      }
      }
      void closeVariables(){
        int maxId = max(id in decVariables) id;
        idsa = new int[id in 0..maxId] = -1;
31      int currentId = 0;
        forall(id in decVariables) idsa[id] = currentId++;

        decVars = 0..nbDecId-1;
        lastModified = new int[decVars] = -1;
36    }
      void close(){
        if ( idsa == null ) closeVariables();
        super.close();
        iter++;
41    }
      void performMove(Closure cl){
        iter := iter + 1;
        call(cl);
      }
46    int getVLSNId(var{int} v) {return idsa[v.getId()];}
      int getVLSNId(var{set{int}} v){
        return idsa[v.getId()];
      }
      range getDecisionVariables()  {return decVars;}
51
      int getModified(var{int} v){
        return lastModified[idsa[v.getId()]];
      }
      int getModified(var{set{int}} v){
56      return lastModified[idsa[v.getId()]];
      }
      void setModified(int id){
        lastModified[idsa[id]] = iter;
      }
61
      Integer getIter(){return iter;}
    }
```

Listing 5.1: The VLSN Solver.

5.2 Getting the Output and Input Variables: the CBVLSN API

The CBLS API has been extended to include input and output variables. A human very easily knows what are the input variables of the differentiable invariants. Our approach is thus based on putting this knowledge directly into the constraints and objective functions.

The output variables are defined in the implementation where the moves are defined (Listing 5.3 and 5.4). A `VarsCollector` collects input and output variables (Listing 5.2).

The CBVLSN API for the differentiable invariants contains methods to retrieve the input variables of the moves, plus the traditional CBLS interface that supports the differentiation of the different supported moves (Listing 5.5 for permutation, and Listing 5.6 for partitionning problems).

The CBVLSN API enables the design of generic VLSN algorithms. Indeed the API is common to all differential invariants; the semantic of the differential invariants is thus hidden behind this API. Then these differential invariants can be generically queried to differentiate atomic moves and to retrieve input and output variables. This allows VLSN search procedures to differentiate a set of moves and to compute independence and compositionality, even if the search procedure has no knowledge about the semantic of the model.

```
interface VarsCollector {
  void clear();
  void addInputVariable(var{int} v);
  void addInputVariable(var{set{int}} v);
  void addOutputVariable(var{int} v);
  void addOutputVariable(var{set{int}} v);
}
```

Listing 5.2: The `VarsCollector` object is used to collect input and output variables.

```
class Permutation{
  var{int}[] perm;
  void assign(var{int} v, int val){ v := val; }
  void swap(var{int} v1, var{int} v2){ v1 :=: v2; }
  void getOutputAssignVariables(var{int} v, int val,
  VarsCollector vc){
    vc.addOutputVariable(v);
  }
  void getSwapOutputVariables(var{int} v1, var{int} v2,
  VarsCollector vc){
```

```
    vc.addOutputVariable(v1);
    vc.addOutputVariable(v2);
  }
}
```

Listing 5.3: Permutation object, with the definition of the moves and the output variables for each of them.

```
class Partition{
  var{set{int}}[] S;
  var{int}[] p;
  dict{int->int} idToIndex;
  Partition(Solver<VLSN> m, int n, int K){
    S = new var{set{int}}[1..K](m);
    p = new var{int}[1..n](m);
    idToIndex = new dict{int->int}();
    forall(k in 1..K) idToIndex{S[k].getId()} = k;
  }
  void insert(var{set{int}} v, int j){
    int k = idToIndex(v.getId());
    S[k].insert(j);
    p[j] := k;
  }
  void getInsertOutputVariables(var{set{int}} v, int j,
  VarsCollector vc){
    vc.addOutputVariable(v);
    vc.addOutputVariable(p[j]);
  }
  void remove(var{set{int}} v, int j){v.delete(j);}
  void getRemoveOutputVariables(var{set{int}} v, int j,
  VarsCollector vc){
    vc.addOutputVariable(v);
  }
  void exchange(var{set{int}} v, int i, int j){
    int k = idToIndex(v.getId());
    S[k].delete(i);
    S[k].insert(j);
    p[j] := k;
  }
  void getExchangeOutputVariables(var{set{int}} v, int i
  , int j, VarsCollector vc){
    vc.addOutputVariable(v);
    vc.addOutputVariable[p[j]];
  }
}
```

Listing 5.4: Partition object, with the definition of the moves and the output variables for each of them.

```
1  interface PermutationDifferentialInvariant<VLSN> {
2    Solver<VLSN> getLocalSolver();
3    var{int} value();
4    int  getAssignDelta(var{int} var,int v);
5    void getAssignInputVariables(var{int} var,int v,
6                                 VarCollector t);
7    int  getSwapDelta(var{int} v1, var{int} v2);
8    void getSwapInputVariables(var{int} v1,var{int} v2,
9                               VarCollector t);
10 }
```

Listing 5.5: The differentiable invariants on permutations can be differentiated wrt the supported moves (Lines 4 and 7). This is part of the standard CBLS API. In addition, in our CBVLSN API, these differentiable invariants can be queried to retrieve the input variables wrt the supported moves (Lines 5 and 8).

```
   interface PartitionDifferentialInvariant<VLSN> {
     Solver<VLSN> getLocalSolver();
     var{int} value();
     int getExchangeDelta(var{set{int}} S,int i,int j);
5    void getExchangeInputVariables(var{set{int}} S,int i,
                                    int j, VarCollector t);
     int getInsertDelta(var{set{int}} S,int i);
     void getInsertInputVariables(var{set{int}} S,int i,
                                  VarCollector t);
10   int getRemoveDelta(var{set{int}} S,int j);
     void getRemoveInputVariables(var{set{int}} S,int j,
                                  VarCollector t);
   }
```

Listing 5.6: CBVLSN API for differentiable invariants on partitions. They contain methods to be differentiated wrt the moves, and methods to retrieve the corresponding input variables.

These variables are used to compute independence and compositionality. The main operations to perform are intersection and union of input and output variables (Definitions 15 and 18). These variables are stored as sets of integers, because the decision variables can be uniquely identified by their integral LS identifiers (Section 5.1.1). These sets are represented by arrays of bits. This allows to compute the intersection and union by performing $\lceil \frac{n}{32} \rceil$ AND and OR operations on integers, where n is the number of decision variables and if we use a 32 bits computer. This representation of input and output variables leads to the object `BitSetCollection` that represents a collection of sets repre-

sented as bit arrays (Listing 5.7).

```
native class BitSetCollection{
  bits(int lb , int ub, int m);
  void insert(int i,  int e);
  void remove(int i,  int e);
  bool areDisjoint(int i, int j);
  void _union(int i, int j);
  void _inter(int i, int j);
  void empty(int i);
  bool contains(int i, int e);
  void copy(int i, int j);
}
```

Listing 5.7: A BitSetCollection represents a collection of `m` bits arrays. These arrays store sets containing elements from `lb` to `ub`. For example the method `insert(i,e)` inserts the element `e` to the set at index `i` in the collection. The method `_union(i,j)` inserts all the elements in the set at index j in the set at index i.

5.3 Describing MoveGraphs

Any MoveGraph $MG = \langle V, E, \eta \rangle$ is described by implementing a few methods (Listing 5.8). An object `MoveGraph` represents a MoveGraph (Definition 8), to use in conjunction with one constraint $\mathcal{C}_2$ and one objective function f. Indeed, in practice a MoveGraph is used with only one constraint and one objective function per program.

The method `getNodes` returns the set of nodes V. In theory, the nodes can be any mathematical object. However for efficiency, the set of nodes V is restricted to be a set of consecutive integers in our implementation. The set of edges E is defined by the method `isMove(i,j)` that returns whether an edge (i, j) is contained in E or not. There is no object representing moves in Comet, and we think that such object would not be efficient to implement and to use in practice. We thus cannot implement the function $\eta : V \to \mathcal{M}$ directly. Five methods implement the operations to be done with a move $\eta(i, j)$:

`applyMove` performs the move $\eta(i, j)$

`isMoveFeasible` returns whether the move $\eta(i, j)$ respects the constraint $\mathcal{C}_2$

`getMoveDelta` returns the differentiation of move $\eta(i, j)$ on the objective function f

`getVariables` that records the input and output variables of the move $\eta(i, j)$

Once these methods are implemented, the MoveGraph is fully described and can be used to compute the differentiation of several moves efficiently and to search VLSN. The interface `MoveGraph` is also extended to describe MoveGraphs to be used when the input and output variables of $\eta(i, j)$ can be expressed as a function of i and j, an explained in Section 4.2.2 (Listing 5.9).

We also give the implementation of the two MoveGraphs for permutation and partitinning problems from Definitions 10 and 11 (Listings 5.10 and 5.11).

```
interface MoveGraph{
  MoveGraph(){}

  range getNodes();
  bool isMove(int i, int j);

  void applyMove(int i, int j);
  bool isMoveFeasible(int i, int j);
  int getMoveDelta(int i, int j);
  void getVariables( int i, int j ,VarsCollector vc);
}
```

Listing 5.8: Description of a MoveGraph.

```
interface NodeMoveGraph extends MoveGraph{
  void getVariables(int i, VarsCollector vc);
  void getVariables(int i, int j, VarsCollector vc){
    getVariables(i, vc);
    getVariables(j, vc);
  }
}
```

Listing 5.9: Description of a MoveGraph to be used with a model such that the input and output variables of the move η_{ij} are function of i and j (Section 4.2.2).

```
class ExchangePermutationMG implements MoveGraph{
  Permutation p;
  PermutationConstraint C;
  PermutationFunction f;
  ExchangePermutationVLSN( Permutation pp,
  PermutationConstraint Cp, Permutation fp){
```

```
      p = pp; C = Cp; f = fp;
    }
    range getNodes(){
      return p.rng();}

    bool dependOnlyOnJ(){return false;}
    bool isMove(int i, int j){
      return true;
    }
    void applyMove(int i, int j){
      var{int} v = p.get(j);
      v := p.get(i);
    }
    void getVariables(int i, int j, VLSNTracker t){
      f.getAssignOutputVariables(p.get(j),p.get(i),t);
    }
    int getMoveDelta(int i, int j){
      return f.getAssignDelta(p.get(j),p.get(i));
    }
    bool isMoveFeasible(int i, int j){
      return C.isAssignFeasible(p.get(j),p.get(i));
    }
}
```

Listing 5.10: Implementation of the MoveGraph describing the exchange cyclic neighborhood for permutations (Definition 10).

```
abstract class UserExchangeVLSN extends NodeMoveGraph{
  Partition p;
  int K;
  int V;

  set{int}[] values;
  int[] k;
  UserExchangeVLSN(Partition pp, int nbNodesp){
    p = pp;

    K = p.getK().getUp();
    V = nbNodesp;

    k = new int[1..V];
    values  = new set{int}[1..V];
  }

  set{int} getValues(int i);

  void update(){
    forall(i in 1..V){
```

```
        values[i] = getValues(i);
        select(e in values[i])
          k[i] = p.getIndex(e);
      }
27    }
      range getNodes(){ return -K+1..V;}

      void getVariables( int j ,VLSNTracker t){
        if (j <= 0)
32        t.write(p.getPartition(-j+1));
        else
          t.write(p.getPartition(k[j]));
      }
      bool isMove(int i, int j){
37      return (i > 0 || j > 0) && k[i] != k[j];
      }
      void applyMove(int i, int j){
        if ( j <= 0 ) {                // Insert
          p.insert(values[i],-j+1);
42      }else if ( i <= 0 ) {          // Remove
          p.remove(values[j]);
        }else {                        // Exchange
          p.exchange(values[i],values[j]);
        }
47    }
      int getMoveDelta(int i, int j){
        if ( j <= 0 ) {
          return f.getInsertDelta(p.getPartition(k[j]),
          values[i]);
        }else if ( i <= 0 ) {
52        return f.getRemoveDelta(p.getPartition(k[j]),
          values[j]);
        }else{
          return f.getExchangeDelta(p.getPartition(k[j]),
          values[i],values[j]);
        }
      }
57    bool isMoveFeasible(int i, int j){
        if ( j <= 0 ) {
          return C.isInsertFeasible(p.getPartition(k[j]),
          values[i]);
        }else if ( i <= 0 ) {
          return C.isRemoveFeasible(p.getPartition(k[j]),
          values[j]);
62      }else{
          return C.isExchangeFeasible(p.getPartition(k[j]),
          values[i],values[j]);
        }
      }
    }
```

Listing 5.11: Implementation of the MoveGraph describing the exchange

cyclic neighborhood for partitions (Definition 11). Here the nodes 1 to V represents subsets of elements that are in the same subset of the partition in the current solution. These subsets are defined by the method `getValues(i)`. Such subsets of elements will be exchanged among the different subsets of the partition. The nodes $-k$ represents the entire subset S_k of the partition.

5.4 Checking Independence and Compositionality

We here describe how we check for the compositionality and independence of a set of moves M.

We assume in the following that the input and output variables of a move $\eta(i, j)$ can be expressed as function of i and j, as it is the case for many problems (Section 5.3). Indeed the implementation of the different components is harder with this assumption; several optimizations can be done. The implementation of the components for the general case (the input and output variables cannot be expressed as functions of i and j) is simpler and won't be described here. For sake of clarity, we also assume the nodes in the MoveGraph range from 1 to n, where n is the number of nodes.

The input and output variables are computed at the beginning of each iteration and stored in a `BitSetCollection`. This object is set to contain $2n + 2$ sets where n is the number of nodes in the MoveGraph. The input variables of node i ($\text{Var}^<(i, \sigma)$) are stored in the $2i^{\text{th}}$ set of the collection, while the output variables of node i are stored in the $2i + 1^{\text{th}}$ set. The sets at indices 0 and 1 are used to compute the union of the input and output variables of several moves.

It is easy to compute whether a set of moves M is compositional and independant once the `BitSetCollection` is built (Listing 5.12). The object `CompoAndIndepChecker` stores a set of moves M. It also maintains the union of the input and output variables of the moves in M in the sets at indices 0 and 1 respectively.

A move $\eta(i, j)$ is added to M by calling the method `mark(i,j)`. This method returns true if and only if $\eta(i, j)$ is independant and compositional with the moves already in M. This method also adds the input and output variables of $\eta(i, j)$ in the sets 0 and 1. The set of moves M and the sets at indices 0 and 1 are emptied by calling the method `unmark()`.

Then we can efficiently compute whether a move m is compositional and independent with the moves in M (method `isCompoAndIndep`).

```
class CompoAndIndepChecker{
  MoveGraph mg;
  BitSetCollection ioVars;
  int[] mod;
  int nbMarked;

  void update(){
    BSCFiller bsc(ioVars, mod);
    forall(n in nodes){
      bsc.fill(n);
      mg.register(n, bsc);
      mod[n] = bsc.getModified();
    }
  }
  bool mark(int i, int j){
    if (nbMarked==0){
      if (    !ioVars.isDisjoint(2*i  , 2*j+1)
           || !ioVars.isDisjoint(2*i+1, 2*j  )
           || !ioVars.isDisjoint(2*i+1, 2*j+1) )
        return false;
      ioVars._union(0, 2*i  );
      ioVars._union(1, 2*i+1);
      ioVars._union(0, 2*j  );
      ioVars._union(1, 2*j+1);
    }else{
      if (    !ioVars.isDisjoint(0, 2*j+1)
           || !ioVars.isDisjoint(1, 2*j  )
           || !ioVars.isDisjoint(1, 2*j+1) )
        return false;
      ioVars._union(0, 2*j  );
      ioVars._union(1, 2*j+1);
    }
    nbMarked++;
    return true;
  }
  void unmark(){
    ioVars.empty(0);
    ioVars.empty(1);
    nbMarked = 0;
    return true;
  }
  bool isCompoAndIndep(int i, int j){
    return ioVars.isDisjoint(0, 2*j+1)
        && ioVars.isDisjoint(1, 2*j  )
        && ioVars.isDisjoint(1, 2*j+1);
  }
}
```

Listing 5.12: Checking independence and compositionality. The entry `mod[i]` stores the last iteration one of the variables in the set `i` of the collection has been modified. The object `BSCFiller` implements the interface `VarsCollector`. It stores the `BitSetCollection` and the array `mod` as instance variables. We indicate which entry of the collection and of `mod` has to be filled (line 10). Then it is passed to the MoveGraph that will call `bsc` for each variable. At each of these calls, the `BSCFiller` will insert the variable in the sets $2n$ or $2n+1$ and will update `mod[n]` if needed.

5.5 Differentiating Several Moves

Our implementation can efficiently differentiate many sets of moves, if they are compositional and independent. First by precomputing the improvement graph at the beginning of each iteration. Second by checking compositionality and independence of each set of moves.

The improvement graph is a MoveGraph where (1) the edges representing moves not respecting the constraint have been removed, and (2) a weight equal to the differentiation of the corresponding atomic moves have been assigned to the edges (Definition 13). The improvement graph $IG = (V, E_{IG}, w)$ is stored by a cost adjancency matrix c, where $c_{ij} = \infty$ if $(i,j) \notin E_{IG}$ and $c_{ij} = w_{ij}$ otherwise.

The Improvement graph can be automatically derived from a MoveGraph (Listing 5.13). It also can be updated incrementally at each iteration (Listing 5.14).

```
class ImprovementGraph{
  MoveGraph mg;
  int[,] cost;

  void buildFromScratch(){
    forall(j in mg.getNodes(), i in mg.getNodes() )
    cost[i,j] =
      (mg.isMove(i,j) && mg.isMoveFeasible(i,j))
      ? mg.getMoveDelta(i,j)
      : System.getMAXINT();
  }
}
```

Listing 5.13: An Improvement Graph is built from the MoveGraph `mg` and is stored by a adjacency matrix `cost`.

```
class IncrementalNodeIG extends ImprovementGraph{
  int lastComputed;
  void incrementalUpdate(int[] mod){
   forall(j in nodes: mod[j] > lastComputed[j],
            i in nodes )
      cost[i,j] =
        (mg.isMove(i,j) && mg.isMoveFeasible(i,j))
        ? mg.getMoveDelta(i,j)
        : System.getMAXINT();

    forall(i in nodes : mod[i]  > lastComputed[i],
           j in nodes : mod[j] <= lastComputed[j] )
        cost[i,j] =
          (mg.isMove(i,j) && mg.isMoveFeasible(i,j))
          ? mg.getMoveDelta(i,j)
          : System.getMAXINT();
    lastComputed = m.getIter();
  }
}
```

Listing 5.14: Improvement Graph that can be updated incrementally. The variable `lastComputed` stores the last iteration this Improvement Graph has been updated. The parameter `mod` is an array indexed by the nodes. The entry `mod[i]` contains the last iteration any input or output variables of node i has been modified. The update is done in two phase. First we update the incident edges to all the nodes j whose variables have been modified. Second we update the incident edges of the nodes j whose variables have not been modified and such that the variables of i have been modified.

It is easy to differentiate a set of moves once the improvement graph, the input and output variables have been precomputed. This differentiation is computed through the object `MoveCompound` (Listing 5.15). It extends `CompoAndIndepChecker` by providing a method `getDelta(i,j)`. This method returns ∞ if $m = \eta(i, j)$ is not compositional or independant with M. Otherwise it returns the weight set to edge (i, j) in the Improvement Graph.

```
class MoveCompound extends CompoAndIndepChecker{
  IncrementalNodeIG ig;
  void update(){
    super.update();
    ig.incrementalUpdate(mod);
  }
  int getCost(int i, int j){return ig.getCost(i,j);}
```

```
    int getDelta(int i, int j){
      if (!isCompoAndIndep(i,j)) return System.getMAXINT()
      ;
      else return ig.getCost(i,j);
    }
  }
```

Listing 5.15: Differentiation of a set of moves. This object is built on top of the checker of compositionality and independence. Note how this differentiation is made easy thanks to the concepts of input output variables and improvement graph.

5.6 Searching the Cyclic Neighborhood

The VLSN search algorithms are efficiently and simply implemented using the above abstractions. First we extended `MoveCompound` to explicitely store the set of moves M and to provide some simple methods easing the writing of the cyclic VLSN search procedure (Listing 5.16). Second the procedure searching for improving compositional and independent cycles is written concisely (Listing 5.17).

The main advantages of this concise procedure are twofold. First it can be easily extended to integrate meta-heuristics (Listings 5.18 and 5.19). Second the procedure in Listing 5.17 is generic wrt the implementation of the MoveCompound. In the context of this thesis we presented an implementation relying on compositionality and independence to efficiently differentiating a set of moves. But the `MoveCompound` could be implemented differently, enabling the computation of set of moves that are not compositional or independent (for example using simulation).

```
  class CyclicMoveCompound extends MoveCompound{
    int[] is;
    int[] js;
    bool[] markedNodes;

    MoveSubset getMarkedMoves(int i, int j, int cost){
      is[nbMarked+1] = i;
      js[nbMarked+1] = j;
      int p;
      for(p=1; is[p] != j;p++){}
      return MoveSubset(is,js,p..nbMarked+1,cost);
    }
    bool isAnEdge(int i, int j){
      return getCost(i,j) < System.getMAXINT();
    }
    bool isMarked(int i){return markedNodes[i];}
```

```
17    bool mark(int iid, int jid){
        if (super.mark(iid,jid)){
          markedNodes[iid] = true;
          markedNodes[jid] = true;
          is[  nbMarked  ] = iid;
22        js[  nbMarked  ] = jid;
          return true;
        }else return false;
      }
      bool unmark(){
27      forall(i in 1..nbMarked) {
          markedNodes[is[i]] = false;
          markedNodes[js[i]] = false;
        }
        super.unmark();
32    }
    }
```

Listing 5.16: The `CyclicMoveCompound` helps writing the cyclic search procedure. In addition to `MoveCompound` it also stores explicitely the set of moves M in the arrays `is` and `js`, $\eta(is[1],js[1])$ is the first move added in M,..., and $\eta(is[nbMarked],js[nbMarked])$ is the last marked move. It also stores which nodes are endpoints of the moves in M (`markedNodes`). These data are maintained when calling the methods `mark` and `unmark`. The method `getMarkedMoves(i,j,c)` is called when the node j is marked. In this case adding the move $\eta(i,j)$ to M would create a cycle (perhaps not containing all the moves in M). This methods returns an object containing all the edges corresponding to this cycle.

```
    function bool markNodes(int i, int[] pred,
2                           CyclicMoveCompound ms){
      return pred[i]>=0 ||
        markNodes(pred[i],pred,ms) && ms.mark(pred[i],i);
    }

7   function searchCycle(CyclicMoveCompound cmc){
      MoveGraph mg = cmc.getIG().getMoveGraph();
      int inf = System.getMAXINT();
      forall(s in mg.getNodes()){
        int d[-1..mg.getNodes().getUp()] =inf;
12      int pred[mg.getNodes()] = -1;
        ILIST LIST = DeQueue(mg.getNodes());

        LIST.put(s,0,0);
        d[s] = 0;
17
```

```
    while( !LIST.isEmpty()){
      int i = LIST.get();
      if ( markNodes(i,pred,cmc) ){
        forall(j in mg.getNodes() : cmc.isAnEdge(i,j)){
22        int cij = cmc.getDelta(i,j);
          if( cmc.isMarked(j) ){
            int cost = d[i] + cmc.getCost(i,j) - d[j];
            return cmc.getMarkedMoves(i,j,cost);
          }else if(    !cmc.isMarked(j) && cij < inf
27                  && d[i]+cij < 0 && d[j] > d[i] + cij
                     ){
            LIST.put(j, d[j], d[i] + cij);
            pred[j] = i;
            d[j] = d[i] + cij;
          }
32      }
      }
      cmc.unmark();
    }
  }
37 }
```

Listing 5.17: Cyclic VLSN search algorithm. Our abstractions allows the concise writing of complex VLSN search procedures. This helps designing and testing many variants of the same algorithm.

```
Solver<VLSN> m();
Partition p(m,n,K);
3 PartitionConstraint C = ...;
PartitionFunction   f = ...;
m.close();

PartitionMoveGraph mg(p,C,f);
8 ImprovementGraph ig(mg);
CycleMoveCompound cmc(ig);

int inf = System.getMAXINT();

13 forall(e in 1..n){
  whenever p.getIndex(e)@changes(){
    select(it in 2..4){
      tabuUntil[e] = m.getIter() + it;
    }}}
18
while(true){
  set{int} nodes = setof(n in mg.getNodes()) (tabuUntil[
  n] <= m.getIter());
  forall(s in nodes){
    int d[-1..mg.getNodes().getUp()] =inf;
```

```
23      int pred[mg.getNodes()] = -1;
        ILIST LIST = DeQueue(mg.getNodes());
        LIST.put(s,0,0);
        d[s] = 0;
        while( !LIST.isEmpty()){
28        int i = LIST.get();
          if ( markNodes(i,pred,cmc) ){
            forall(j in nodes : cmc.isAnEdge(i,j)){
              int cij = cmc.getDelta(i,j);
              if( cmc.isMarked(j) ){
33              int cost = d[i] + cmc.getCost(i,j) - d[j];
                return cmc.getMarkedMoves(i,j,cost);
              }else if(    !cmc.isMarked(j) && cij < inf
                           && d[i]+cij < 0 && d[j] > d[i] +
                           cij ){
                LIST.put(j, d[j], d[i] + cij);
38              pred[j] = i;
                d[j] = d[i] + cij;
              }}}
          cmc.unmark();
        }}}
```

Listing 5.18: Cyclic VLSN search algorithm with a tabu meta-heuristic (for partitionning problems). The array `tabuUntil` is indexed by the elements of the partition, and stores the iteration until elements cannot be moved from their current partition. The set of nodes representing elements that are not tabu for the current iteration is computed (`nodes`), and the two `forall` loops of the cyclic VLSN search procedure are modified to only loop on these nodes.

```
Solver<VLSN> m();
Partition p(m,n,K);
3 PartitionConstraint C = ...;
PartitionFunction   f = ...;
m.close();

PartitionMoveGraph mg(p,C,f);
8 ImprovementGraph ig(mg);
CycleMoveCompound cmc(ig);
int inf = System.getMAXINT();

UniformDistribution u(1..10000);
13 float t=30;

while(true){
  set{int} nodes = mg.getNodes();
  forall(s in nodes){
18   int d[-1..mg.getNodes().getUp()] =inf;
```

```
      int pred[mg.getNodes()] = -1;
      ILIST LIST = DeQueue(mg.getNodes());
      LIST.put(s,0,0);
      d[s] = 0;
23    while( !LIST.isEmpty()){
        int i = LIST.get();
        if ( markNodes(i,pred,cmc) ){
          forall(j in nodes : cmc.isAnEdge(i,j)){
            int cij = cmc.getDelta(i,j);
28          if( cmc.isMarked(j) ){
              int cost = d[i] + cmc.getCost(i,j) - d[j];
              return cmc.getMarkedMoves(i,j,cost);
            }else if(    !cmc.isMarked(j) && cij < inf
                         && d[j] > d[i] + cij
33                       && u.get() < exp^(-1.0*(d[i]+cij)
                         /t) < 0 ){
              LIST.put(j, d[j], d[i] + cij);
              pred[j] = i;
              d[j] = d[i] + cij;
            }}}
38      cmc.unmark();
      }}
    update t;
  }
```

Listing 5.19: Cyclic VLSN search procedure with simulated annealing. A path is extended by an edge (i,j) if its length is negative. Otherwise it is extended with a probability proportional to its length (line 33).

Part III

Applications

This section illustrates how our new concepts allow the direct implementation of sophisticated VLSN search algorithms from the literature and how they are a key tool to prove them. The experimental results presented here are a proof of concept of our theoretical framework.

First the Capacitated Minimum Spanning Tree problem is modelled and searched using our generic approach. This allows to compare the efficiency of our work compared to a dedicated C++ implementation of VLSN search algorithm.

Second, the Capacitated Exam Timetabling Problem is solved. This allow to illustrate the benefits of the notion of compositionality that is novel in this work.

Finally, the Vehicle Routing with Time-Windows is considered and we illustrate how complex VLSN search algorithms can be implemented and extended using our approach.

Chapter 6

The Capacitated Minimum Spanning Tree

The Capacitated Minimum Spanning Tree (CMST) is a communication problem. An efficient VLSN implementation in C++ has been developed in the literature to solve this problem [AOS03]. *This section demonstrates that the efficiency of our generic approach is comparable to a dedicated implementation in C++.*

The CMST is a partitionning problem and its formulation is similar to the Generalized Assignment Problem (GAP) described in Section 2; only the objective functions are different. Thus, once a CBVLSN algorithm has been developped for the GAP, most of its parts can be reused to solve the CMST using VLSN. This motivates the design of a constraint-based approach.

A VLSN for the CMST was proposed in [AOS01, AOS03] and it finds the best known solutions to this problem. It uses a cyclic neighborhood and searches for cycles passing through any subset at most once. Their neighborhood is equivalent to our compositional cyclic neighborhood but is hardcoded in their search algorithm. It is thus easy to implement their algorithm by means of our constraint-based framework.

6.1 Problem Description

The Capacitated Minimum Spanning Tree problem is about partitionning a set of n terminals into K subnetworks. We are given a matrix $d \in \mathbb{R}^{n+1 \times n+1}$ specifying the distance between all pairs of the n terminals and between all these terminals and a particular node R called the root. Each terminal i also requires a given bandwidth b_i. The goal of the CMST is to find a partition $\mathcal{X} = [S_1, \ldots, S_K]$ of the n terminals,

respecting the capacity constraint

$$\sum_{i \in S_k} b_i \leq D \qquad \forall k = 1, 2, \ldots, K$$

and minimizing the following objective function

$$f_{CMST} = \sum_{k=1}^{K} \left(mst(S_k, d) + \min_{i \in S_k} d_{R,i} \right)$$

where $mst(S_k, d)$ computes the cost of the minimum spanning tree over the terminals in S_k given the cost matrix d. The CMST is modeled as the COP $\langle f_{CMST}, \mathcal{C}_{part} + \mathcal{C}_{capa}, \mathcal{X}, 2^N \rangle$, where $N = \{1, \ldots, n\}$, $\mathcal{C}_{part}$ is the global partition constraint and $\mathcal{C}_{capa}$ is the capacity constraint, as for the GAP. We refer the reader to [AOS01, AOS03] for extensive references about the CMST.

6.2 A Concise Model

The CMST can be modelled as depicted in Listing 6.1. The variable `p` represents a partition of the elements $\{1, \ldots, n\}$ into K sets. The differentiable objective `mst` computes the cost of the minimum spanning trees for a set of nodes, and `Obj` is a differentiable invariant representing the cost of a partition for this problem. The constraint system `Cs` contains all the constraints $\mathcal{C}_2$ of the problem, i.e. the capacity constraints in this case. The input and output variables wrt the model of the GAP and the moves defined on partitions have been described in Example 10. In our implementation, the input and output variables are defined only for each separate differential invariant.

```
Solver<VLSN> vs();
Partition<VLSN> p(vs,1..n,1..K);
var{set{int}}[] S = p.getSubsets();
PartitionFunction<VLSN> Obj(
   sum(k in 1..K) (mst(vs,S[k],d) + min(i in S[k]) d[R,i
   ]));
PartitionConstraintSystem<VLSN> Cs(m);
Cs.post(CapacityConstraint(p,b,B));
vs.close();
```

Listing 6.1: Model for the CMST

6.3 An Efficient Search

In order to compare our approach to the dedicated implementation described in [AOS01], we reproduced their VLSN search procedure using our framework. We used a Greedy Randomized Adaptative Search Procedure (GRASP). The principle of such procedure is to compute an initial solution using a randomized greedy algorithm, and then improving it by performing VLSN moves. When no improving move can be identified, the process in restarted. The search algorithm stops when a given time limit is reached.

In order to fully describe a GRASP, we thus need to explain how the initial solutions are computed, and which VLSN search algorithm is used to improve them. We compute the initial solution as in [AOS01] by using a randomized version of the greedy algorithm proposed in [EW66]. This algorithm starts with each subtree containing a singleton node. In each iteration, the algorithm joins two subtrees into a single subtree so that the new subtree satisfies the capacity constraints and the savings achieved by the join operation are maximum. We select randomly one of the three join operations achieving the bests savings exactly as it is done in [AOS01].

These initial solutions are improved using the cyclic neighborhood. The search first looks for an improving neighbor in the cyclic exchange neighborhood. If such a move exists, it is applied. Otherwise, a new randomized initial solution is computed and the search restarts. This search is depicted in Listing 6.2. Line 1 constructs the move graph and line 2 automatically derives its associated improvement graph. In line 4, the best move found by the cyclic search algorithm is obtained: it encapsulates the actual move and the improvement graph (and thus the move graph) to update the improvements when applied. It also can be differentiated as shown in line 5. Observe how the search is completely separated from the model and could thus be used for any other partitioning problems without any modification.

6.4 Enlarging the Neighborhood

The VLSN algorithm presented above only exchanges single values between set variables. In [AOS03], a more complex neighborhood is presented, in which subtrees of the current solution and/or single values can be exchanged among different subsets as illustrated in Figure 6.1. Our abstractions allow to implement this new neighborhood by simply extending the MoveGraph presented in Section 4.1.3. There is no need

```
  Solution bestSol(m);
2 int bestCost = Obj.value();
  PartitionExchangeMoveGraph mg(vs,p,Obj,Cs);
  ImprovementGraph ig(mg);
  CycleMoveCompound cmc(ig);
  while (System.getCPUTime() < timeLimit ){
7   MoveSubset M = cmc.getBestCycle();
    if (M == null || M.getDelta() > 0) p.initialize();
    else M.apply();
    if (bestCost > Obj.value()){
      bestCost = Obj.value();
12    bestSol = Solution(m);
    }
  }
```

Listing 6.2: Search for the CMST

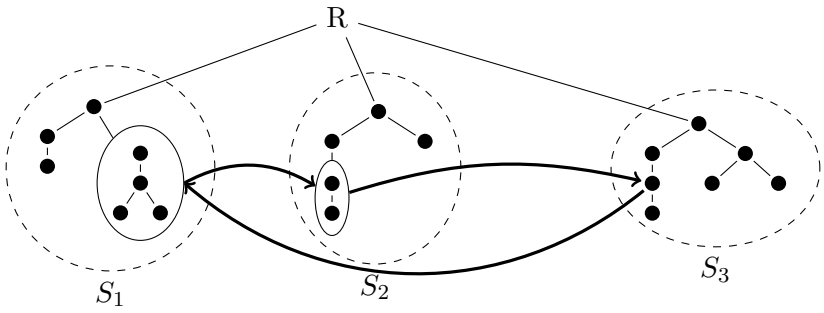


Figure 6.1: Composite Cyclic Exchange for the CMST.

to modify the model or the search algorithm provided that the CBLS API supports the atomic moves.

This complex neighborhood can be implemented as depicted in Listing 6.3.

```
1 class CMSTMoveGraph extends PartitionExchangeMoveGraph{
    set{int} getValues(int i){
      return (i > n)
        ? getSubtree(i)
        : {i};
6   }
    set{int} getSubtree(int i){
      //computes the elements in the subtree rooted
      // at i of the MST induced by the current partition.
    }
```

```
11 }
```

Listing 6.3: Class defining a MoveGraph for the CMST problem. A node $i : 1 \leq i \leq n$ represents the element $i \in \{1, \ldots, n\}$, a node $i : n < i \leq 2n$ represents the subtree rooted at the element i in the current solution and a node $i \leq 0$ represents the subset S_{-i+1}.

6.5 Comparing to Dedicated Solutions

The extensive computational study in [AOS01, AOS03] allows us to compare our approach to a dedicated and highly efficient implementation in C++. We reproduced the same analysis as in [AOS01] and on the same instances. The initial solutions are computed exactly as in [AOS01] and the GRASP (Greedy Randomized Adaptive Search Procedure) is tuned with the same parameters (Time limit of 200 seconds, application of the first improving cycle found). The experimental analysis presented in [AOS01] (denoted Du in Table 6.1) was made on a Pentium 1,4 GHz with 512MB of memory. Our algorithm (denoted MDVH) is run on a single core of a machine with an Intel Core Quad CPU Q6600 at 2.4GHz with 1GB memory. The difference of speed between both setups was estimated at a factor 3.3 after running some tests.

Table 6.1 shows that both implementations behave similarly: they compute solutions of equivalent quality (although our results are slighly inferior which may be due to implementation details that were not described in [AOS01]) and perform roughly the same number of iterations per run. The last column shows our implementation is about 4 times slower than the dedicated implementation of [AOS01] (the difference of computers used is already taken into account in Table 6.1), which is not surprising since our abstractions are built on top of Comet. Supporting the abstractions directly in the core of Comet will remove this gap.

Instance	Q	Avg value		Nb of iter		Time per iter		Time
		MDV	Du	MDV	Du	MDV	Du	Factor
tc80-1	5	1112	1108	21.25	23	34.15	20	5.63
tc80-3	5	1087	1082	17.94	18.9	34.73	20	5.73
tc80-5	5	1301	1301	18.94	20.2	34.28	20	5.66
tc80-1	10	905	905	12.69	12	49.61	60	2.73
tc80-3	10	898	890	9.77	7	49.53	70	2.33
tc80-5	10	1036	1023	11.06	6.3	49.19	80	2.03
te80-1	5	2556	2555	15.68	11.6	39.01	30	4.29
te80-3	5	2636	2624	17.4	19.7	35.99	30	3.96
te80-5	5	2491	2486	13.89	18	35.98	20	5.94
te80-1	10	1717	1701	11.23	14.7	52.56	60	2.89
te80-3	10	1731	1719	13.54	13.9	51.94	60	2.86
te80-5	10	1662	1651	10.6	13.4	54.43	60	2.99

Table 6.1: Experimental comparison of our implementation with [AOS01]. Q is the maximum allowed number of terminals in a subset (capacity constraint). *Avg value* is the average of the values found after each run, *Nb of iter* is the average number of iterations per run and *Time per iter* is the average time in milliseconds to find a cycle. The last column indicates the time factor between both implementations, with the difference in computers taken into account.

Chapter 7

The Capacitated Examination Timetabling Problem

This chapter presents a CBVLSN program for solving a hard problem encountered nowadays in universities. It illustrates that compositionality is essential for efficiently driving the search towards good solutions, and consequently to obtain new state-of-the-art solutions.

7.1 Problem Description

The Capacitated Examination Timetabling Problem (CETP) is a real-life – and very complex – problem encountered in universities. The goal of the CETP is to partition n exams into K consecutive time slots subject to an exclusion constraint: there are no students taking two exams scheduled in the same time slot.

The adaptability and reusability of the algorithms solving some variants of this problem is valuable. Indeed each university has its own additional requirements and, depending on its own reality, it may be willing to consider additional constraints or objective. Constraint-based approaches are designed with this challenge in mind.

Here we focus on a variant that has been considered multiple times in the literature [AAB$^+$07, AABD04, AABD07]. There is an additional constraint stating that for each time slot k, the total number of students having an exam scheduled at k is less than a total room capacity D. The objective is to minimize the number of students having two exams the same day in two consecutive time slots. We let $S = [S_1, \ldots, S_K]$ be K set variables, where S_k represents the set of exams scheduled at the

timeslot k for $k = 1, \ldots, K$. The objective function can be formulated as follows:

$$f_{\text{CETP}} = \sum_{\substack{1 \leq k < K \\ day_k = day_{k+1}}} \sum_{\substack{i \in S_k \\ j \in S_{k+1}}} c_{ij} \tag{7.1}$$

where the number of students taking exams i and j is given by the matrix $(c_{ij}) \in \mathbb{N}^{n \times n}$, and day_k is the day of time slot k. The capacity constraint is equivalent to the constraint of the Capacitated Minimum Spanning Tree. The violation of the exclusion constraint can be computed as follows

$$\mathcal{C}_{excl} = \sum_{k=1}^{K} \left(\sum_{i,j \in S_k : i < j} c_{ij} \right) \tag{7.2}$$

This constraint is respected if $c_{ij} = 0,\ \forall i, j \in S_k,\ k = 1, \ldots, K$.

Thus the CETP can be defined as the OCSP $\langle f_{\text{CETP}}, \mathcal{C}_{part} + \mathcal{C}_{capa} + \mathcal{C}_{excl}, S, 2^N \rangle$, where $\mathcal{C}_{part}$ is the global partition constraint, $\mathcal{C}_{capa}$ is the capacity constraint, and $N = \{1, \ldots, n\}$.

This problem can be implemented using our framework as follows.

7.2 Model and Search

The model of the CETP is similar to the CMST, except that there is an additional exclusion constraint to ensure there is no conflict for any student. The COMET model for this problem is illustrated in Listing 7.1 and is very similar to Listing 6.1; observe that the capacity constraint implemented for the CMST can be reused as is for the CEPT.

```
Solver<VLSN> m();
Partition<VLSN> p(m,1..n,1..K);
CETPFunction f(p,day);
PartitionConstraintSystem<VLSN> Cs(m);
Cs.post(CapacityConstraint(p,b,D));
Cs.post(ExclusiveConstraint(p,c));
vs.close();
```

Listing 7.1: A declarative model of the Capacitated Examination Timetabling Problem. Each component of the model captures essential substructure of the problem, and can be queried for differentiability and input variables. Generic VLSN search components can communicate with such models through the CBVLSN API, and are then able to efficiently find very good solutions.

The GRASP search used for the CMST will be reused here (Listing 6.2). This illustrates the reusability of our approach,

7.3 Compositionality Enhances the Search

Several works solved the Capacitated Exams Timetabling Problem by designing a dedicated VLSN approach and obtained the best known solutions to some of these instances [AAB+07, AABD04, AABD07].

These VLSN search algorithms often compute useless iterations; the selected set of moves is often rejected. Indeed they all search for an independent cycle (i.e., a cycle with no pair of moves modifying the same set variable), and do not consider compositionality. Hence they may find a cycle with a negative cost, that is not reflecting the true variation on f_{CETP} of the corresponding moves. In such cases the selected set of moves may be degrading and thus be rejected *a posteriori*. This makes unvaluable the previous work of selecting the negative cycle.

This problem is avoided by considering compositionality. Indeed from Equation (7.1), the variation on the objective f_{CETP} of a move modifying the set variable S_k may depend on the set variables S_{k-1} and S_{k+1}. Compositionality captures this fundamental structure of the objective function. So the cost of a compositional cycle reflects the true variation on f_{CETP} of the corresponding moves. This allows to directly compute a cycle leading to a better solution.

Hard instances coming from Canadian universities are available [CL96] and were considered in [AAB+07, AABD04, AABD07]. Their approach is equivalent to ours, except that they do not check for compositionality during the search for an improving cycle.

We quantified the added value of the novel concept of compositionality. We made 50 runs of our Comet program of 10 minutes, with and without the check for compositionality. The time limit has been chosen such that both algorithms were able to perform enough iterations to illustrate their behavior. We report the average of the cost of the best solution found after a given number of iterations (Figure 7.1).

These results indicate that the notion of compositionality may significantly improve the results of VLSN algorithms. In constraint-based VLSNs, the shortest-path algorithm is driven towards cycles that improve the current solution. If compositionality is not checked, some improving cycles do not necessarily reflect the true variation on the objective function. This may drive the search algorithm towards degrading solutions and reduce the efficiency of the algorithm. This inequality between the

cost of a cycle in the improvement graph and the true variation of the objective wrt the corresponding moves leads to an unwanted random behavior. In constraint-based VLSN, this issue is avoided entirely and compositionality comes from free since it is uderived compositionally from primitive constraints and objectives.

7.4 New state-of-the-art solutions

We found new state-of-the-art solutions to the CETP with our GRASP search by computing better initial solutions and making our algorithm run until no improving cycle is found. We have already described what our COMETprogram does, except how the initial ramdomized solutions are computed. Given an exam i, let CE_i be the set of exams conflicting with i: $CE_i = \{j \in \{1, \ldots, n\} | c_{ij} > 0\}$. In order to assign the exams to the timeslots, we select an exam not already assigned having the largest number of assigned conflicting exams. We break ties by choosing the exam with the most conflicting students $\sum_{j \in CE_i} c_{ij}$, or with the most conflicting exams. We thus select the exam i with the greater lexicographical value

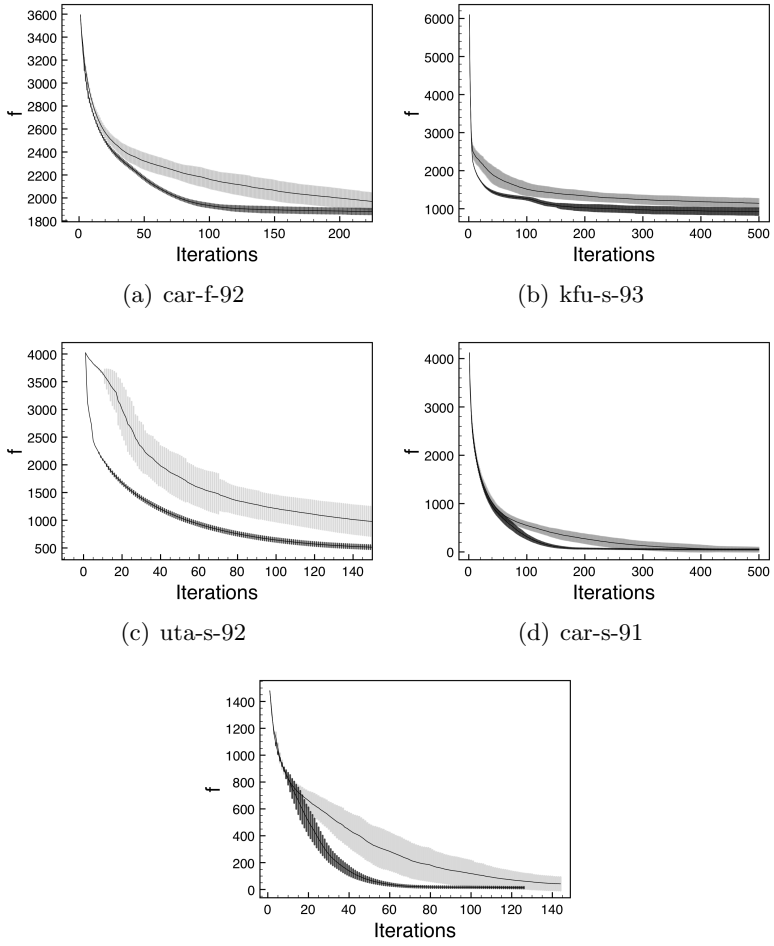
$$\left\langle |\{j \in CE_i | j \text{ is assigned}\}|, \sum_{j \in CE_i} c_{ij}, |CE_i| \right\rangle$$

We then assign the selected exam to the timeslot minimizing the impact on the objective function according to the already assigned exams. We repeat this process until all the exams are assigned. This randomized procedure is not always able to produce a complete partition; an exam can be impossible to assign to any timeslot. However after a few trials, a good initial solution can be found.

Our concise CBVLSN model and search for the CETP improves the best solution found for two of the data sets considered in [AABD07], and obtains the optimal solution for the data set TRE-S-92 (Table 7.1). We compared our algorithm to [AABD07] and [MBHS03] that are the two algorithms computing the best known solutions on these data sets. Unfortunately our procedure computing the initial solutions didn't find a feasible initial partition for the fifth instance CAR-F-92.

Selecting many moves at each iteration is crucial to improve the generated initial solutions. Indeed our algorithm had to explore cycles with many edges in order to identify some improving cycles (max and avg cycle denotes the maximum and average length of the improving cycles returned by our algorithm) .

Our algorithm also performs faster than the two other algorithms.



(a) car-f-92

(b) kfu-s-93

(c) uta-s-92

(d) car-s-91

(e) tre-s-92

Figure 7.1: Experimental results for the CEPT, using a saturation degree heuristic to compute the initial solutions. The efficiency of the cyclic neighborhood without the check for compositionality (brighter curve [AABD07]) is compared to our compositional neighborhood (darker curve). Each curve represents the average of the cost of the best solution found among the fifty runs, after a given number of iterations. The area around the curve represents the standard deviation among the fifty runs. These results illustrate that compositionality enhances the search, by guiding it towards good solutions. Not checking compositionality leads to an unwanted random behavior (illustrated by the larger error area around the bright curve).

Instance		[MBHS03]	AAB[AABD07]	CBVLSN
CAR-S-91				
	Best	31	47	**14**
	Average	47	-	35.05
	Time(sec)	125	overnight	20.2
	# iter	-	-	3.4
	max cycle	-	-	18
	avg cycle	-	-	6.56
KFU-S-93				
	Best	237	**206**	542
	Average	290.6	-	614.75
	Time(sec)	45	overnight	29.8
	# iter	-	-	31.75
	max cycle	-	-	11
	avg cycle	-	-	4.33
TRE-S-92				
	Best	**0**	4	**0**
	Average	0.4	-	0.6
	Time(sec)	16	overnight	1.2
	# iter	-	-	1.2
	max cycle	-	-	10
	avg cycle	-	-	6.08
UTA-S-92				
	Best	334	310	**288**
	Average	393.4	-	347.15
	Time(sec)	173	overnight	29.4
	# iter	-	-	13.85
	max cycle	-	-	19
	avg cycle	-	-	6.47

Table 7.1: Our CBVLSN algorithm improves two instances, and finds the optimal solution for a third one. Our approach is also faster. Howerer it performs poorly on the instance KFU-S-93. The maximum and average length of the improving cycles found by our algorithm are high. This indicates that being able to perform a huge number of moves at the same iteration is crucial to be able to improve the generated initial solutions.

Chapter 8

Vehicle Routing Problem with Time-Windows

This chapter describes a CBVLSN algorithm that finds the best solutions to the Vehicle Routing Problem with Soft Time-Windows (VRPSTW). This problem is an extension of the classical vehicle routing problem with capacity constraints; when the customers should be served is constrained in time. These constraints can be both treated as hard (they absolutely have to be satisfied), or soft (a penalty cost is incurred if they are not satisfied).

New complex problems can be modelled in our framework. As illustrated in this chapter, new constraints and objectives can be quickly designed. This enriches the expressiveness of the framework while existing components can be reused simultaneously. Our framework is open and modular.

Our VLSN search abstractions also perform efficiently on problems with complex objectives. Indeed, the search procedures presented in this thesis can be easily modified to take into account some human knowledge of how to search for good solutions. Moreover, the search procedures described here below can be used on many different problems because the independence and compositionality of moves can be automatically detected wrt a given model.

Developing new constraints, objectives or new search procedures is quick thanks to the provided high-level abstractions. This allows to use our framework to develop state-of-the-art VLSN approaches in industrial environments where short development times are required.

Our algorithm obtains better solutions than previous work on this problem and manages to reduce the number of vehicle used for almost half of the instances. This section illustrates three qualities of our ap-

proach that are important in modern combinatorial optimization: flexibility, efficiency and robustness.

In Section 8.1, the VRPSTW will be defined thoroughly. The model and the search for solving the VRPSTW are described in Sections 8.2 and 8.3, while the implementation is depicted in Section 8.4. The benefits of our approach are assessed in the experimental Section 8.5.

8.1 Definition of the Real-Life Problem

The capacitated vehicle routing problem (CVRP)

Let a fleet of K identical **vehicles**: $Vehicles = \{v_1, \ldots, v_K\}$. Each vehicle has a limited capacity Q.

Let a set of n **customers** $Customers = \{1, \ldots, n\}$ and a central depot D. For ease of notation, we identify the depot as K dummy sites: $Depots = \{n+1, \ldots, n+K\}$. The set $Sites = Depots \cup Customers$ identifies all the sites considered in this problem. The travel cost between sites i and j is denoted c_{ij}. Each customer i has a specific demand q_i.

A **route** r is a sequence of sites $\langle r_0, \ldots, r_{|r|} \rangle$ such that $r_1, \ldots, r_{|r|} \in Customers$ and $r_0 \in Depots$. Such a route represents a tour that starts from the depot, visits a sequence distinct of customers and returns to the depot. The set of customers served by a route r is denoted $cust(r) = \{r_1, \ldots, r_{|r|}\}$ and we let $|r| = |cust(r)|$. Given a route $r = \langle r_0, r_1, \ldots, r_{|r|} \rangle$, we let the successor visit of visit r_i be r_{i+1}. As a route represents a tour from and to the depot, we let the successor of visit $r_{|r|}$ be r_0. We define the subsequent visits of r_i as all the customers visited after r_i: $\{r_{i+1}, \ldots, r_{|r|}\}$. A route r is valid wrt the capacity constraint if $\sum_{i \in cust(r)} q_i \leq Q$.

A **routing** R is a collection of valid routes $R = [r^{(1)}, \ldots, r^{(K)}]$ assigning a valid route $r^{(k)}$ to each vehicle $v_k \in Vehicles$ and such that the routes represent a partition of the set of customers[1].

As a customer i is contained in exactly one route in a routing $R = [r^{(1)}, \ldots, r^{(K)}]$, we define the successor i^+ of visit i as the site visited right after i. A vehicle v_k is used if its route contains at least one customer ($|cust(r^k)| > 0$). We denote the number of used vehicles in a routing R by $|R|$ and the total distance of R as $dist(R) = \sum_{i \in Sites} c_{i,i^+}$.

The CVRP calls for a valid routing wrt the capacity constraint and that minimizes the lexicographical objective $\langle |R|, dist(R) \rangle$.

[1] (1) $cust(r^{(i)}) \cap cust(r^{(j)}) = \emptyset, \forall v_i \neq v_j \in Vehicles$, and (2) $\cup_{v_i \in Vehicles} cust(r^{(i)}) = Customers$.

The vehicle routing problem with Time-Windows (VRPTW)

The vehicle routing problem with Time-Windows is an extension of the CVRP where customers have specified the periods they can be served. A time-window for customer i is an interval $[e_i; l_i]$ where e_i is the earliest service time and l_i is the latest service time ($e_i < l_i$). A time-window can be hard (the customer has to be served within his time-window) or soft (a unit penalty is incurred if the customer is not served within his time-window). We are also given service duration d_i representing the duration of visit to customer i. The feasibility and the penalty of the soft time-windows constraints of a routing R depends on the service time s_i which has to be assigned to each customer i. From the service times, we can derive the departure time of customer i:

$$\delta_i \geq 0 \qquad \forall i \in Depots \tag{8.1}$$

$$\delta_i = s_i + d_i \qquad \forall i \in Customers \tag{8.2}$$

Given a routing, because the vehicle must spend some time at visit i and must travel from this visit to its successor, the service times must satisfy the constraints

$$\delta_i + c_{i,i^+} = s_{i^+} \qquad \forall i \in Depots \tag{8.3}$$

$$\delta_i + c_{i,i^+} \leq s_{i^+} \qquad \forall i \in Customers \tag{8.4}$$

Service times $\{s_i | i \in Sites\}$ are valid if they respect constraint (8.1)-(8.4).

Hard Time-windows The availability of a customer i is specified by the time-window $[e_i; l_i]$. A vehicle can arrive at a site i before e_i but cannot wait more than w_{max} minutes before serving customer i (it should arrive at i after $e_i - w_{max}$). Given a routing R, the problem is to assign valid service times s_i for each customer i such that

$$e_i \leq s_i \leq l_i \tag{8.5}$$

$$s_i + d_i + c_{i,i^+} \geq \&s_{i^+} - w_{max} \tag{8.6}$$

Constraints (8.6) states that the vehicle does not have to wait more than w_{max} before serving a client. A route r is valid wrt the hard time-windows constraint iff there exists valid service times for all $i \in cust(r)$ respecting constraints (8.5) and (8.6).

Because a vehicle needs to depart from the depot and to travel to a customer i before serving him, we assume that

$$e_i \leq c_{0,i} \qquad \forall i \in Customers \tag{8.7}$$

Note that Equation (8.1) is implied by Equation (8.5) and (8.7). Given a valid routing R, for all customer i, the customers visited after i determines an earliest delivery time y_i for i; if the vehicle serves i before y_i, it will have to wait more than w_{max} minutes at one point in the route. These visits also determine a latest delivery time z_i for customer i; if the vehicle serves i later than z_i, then it will be impossible to serve all the subsequent visits before their latest service time. The earliest and latest service time are defined recursively as follows.

$$z_i = \begin{cases} l_i & \forall i \in Depots \\ \min(l_i, z_{i^+} - c_{i,i^+} - d_i) & \forall i \in Customers \end{cases} \tag{8.8}$$

$$y_i = \begin{cases} e_i & \forall i \in Depots \\ \max(e_i, y_{i^+} - w_{max} - c_{i,i^+} - d_i) & \forall i \in Customers \end{cases} \tag{8.9}$$

Proposition 11. *A route $r = \langle r_0, \ldots, r_K \rangle$ is valid wrt the hard time-windows constraint if and only if $y_i \leq z_i$ and $y_i \geq c_{0,i}, \forall i \in cust(r)$.*

Proof. ($\Rightarrow$) From the definitions of the earliest and latest service time, we clearly must have $y_i \leq s_i \leq z_i$ for all $i \in cust(r)$. Thus if $\exists i \in cust(r) | y_i > z_i$, then the route is not valid wrt the time-windows constraint.

($\Leftarrow$) We will prove that if $y_i \leq z_i, \forall i \in Sites$, then there exists a set of service time $\{s_i | i \in cust(r), y_i \leq s_i \leq z_i\}$ such that Equations (8.4) to (8.6) are verified. This can be proven by induction on the customers of a route using the following inductive hypothesis.

Hypothesis 1 (*Recur*(k))**.** *If $y_i \leq z_i, \forall i \in Sites$, then for any service time s_{r_k} such that $y_{r_k} \leq s_{r_k} \leq z_{r_k}$, $\exists [s_i | i \in r_{k+1}, \ldots, r_K, y_i \leq s_i \leq z_i]$ such that $[s_{r_k}, s_{r_{k+1}}, \ldots, s_{r_K}]$ respect Equations* (8.4) *to* (8.6) *for all $i = r_k, \ldots, r_K$.*

Proposition 11 is equivalent to *Recur*(1). We will prove that *Recur*(k) holds for $k = K, \ldots, 1$.

Initial case (k = K): Consider the subroute $\langle r_K \rangle$ and let s_{r_K} such that $y_{r_k} \leq s_{r_k} \leq z_{r_k}$ and $s_{r_0} = \delta_{r_K} + c_{r_K,r_0} = s_{r_K} + d_{r_K} + c_{r_K,r_0}$. Equation (8.4) is clearly respected for $i = r_K$ and we have

$$e_{r_K} \leq y_{r_K} \leq s_{r_K} \leq z_{r_K} \leq l_{r_K}$$
$$e_{r_0} \leq e_{r_K} \leq s_{r_K} + d_{r_K} + c_{r_K,r_0} \leq z_{r_K} + d_{r_K} + c_{r_K,r_0} \leq z_{r_0} \leq l_{r_0}$$

Thus Equation (8.5) is respected for $i = r_K$ and $i = r_0$. And finally

$$s_{r_K} + d_{r_K} + c_{r_K,r_0} = s_{r_0} \geq s_{r_0} \geq s_{r_0} - w_{max}$$

Recursive case: Let suppose $Recur(k+1)$ holds. We prove that $Recur(k)$ holds. Let suppose $y_i \leq z_i, \forall i \in Sites$ and let s_{r_k} such that $y_{r_k} \leq s_{r_k} \leq z_{r_k}$. We have

$$s_{r_k} + d_{r_k} + c_{r_k,r_{k+1}} \leq z_{r_k} + d_{r_k} + c_{r_k,r_{k+1}} \leq z_{r_{k+1}} \quad \text{from (8.8)} \quad (8.10)$$

$$y_{r_{k+1}} \leq z_{r_{k+1}} \quad (8.11)$$

$$y_{r_{k+1}} \leq y_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} + w_{max}$$

$$\leq s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} + w_{max} \quad \text{from (8.9)} \quad (8.12)$$

$$s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} \leq s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} + w_{max} \quad (8.13)$$

Thus

$$[\max(y_{k+1}, s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}}); \min(z_{r_{k+1}}, s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} + w_{max})] \neq \emptyset$$

Let $s_{r_{k+1}} = s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}}$ in this interval. This service time is such that $y_{r_{k+1}} \leq s_{r_{k+1}} \leq z_{r_{k+1}}$ and from $Recur(k+1)$, let $[s_i | i \in r_{k+2}, \ldots, r_K, y_i \leq s_i \leq z_i]$ be such that $[s_{r_{k+1}}, \ldots, s_{r_K}]$ respect Equations (8.4) to (8.6) for all $i = r_{k+1}, \ldots, r_K$.

We then have

$$e_{r_{k+1}} \leq y_{r_{k+1}} \leq \max(y_{k+1}, s_{r_{k+1}}) \quad (8.14)$$

$$s_{r_{k+1}} + d_{r_k+}c_{r_k,r_{k+1}} \leq \ldots \quad (8.15)$$

and

$$\min(z_{r_{k+1}}, s_{r_{k+1}} + w_{max}) \leq z_{r_k+1} \leq l_{r_{k+1}} \quad (8.16)$$

$$\leq s_{r_{k+1}} + w_{max} \quad (8.17)$$

So $[s_{r_k}, s_{r_{k+1}}, \ldots, s_{r_K}]$ respect Equations (8.4) to (8.6) also for $i = r_{k+1}$. This shows that for all $s_{r_k} | y_{r_k} \leq s_{r_k} \leq z_{r_k}, \exists [s_i | i \in r_{k+1}, \ldots, r_K, y_i \leq s_i \leq z_i]$ such that $[s_{r_k}, \ldots, s_{r_K}]$ respect Equations (8.4) to (8.6) for all $i = r_1, \ldots, r_{K+1}$. □ □

The proof of Proposition 11 illustrates how the service times can be determined such that the hard time-windows constraint is respected.

Corollary 1. *Given a route $r = \langle r_1, \ldots, r_K \rangle$, the following method determines valid service times $\{s_i | i \in cust(r)\}$ wrt the hard time-windows constraint:*

1) select a service time $s_{r_1} \in [y_{r_1}; z_{r_1}]$, and then

2) select service times s_{k+1} successively for $k = 1, \ldots, K-1$ in the interval

$$[\max(y_{k+1}, s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}}); \min(z_{r_{k+1}}, s_{r_k} + d_{r_k+}c_{r_k,r_{k+1}} + w_{max})]$$

Soft time-windows We are also given preferences for each customer about the time they would like to be served. These preferences are specified by soft time-windows $[e_i^*; l_i^*]$ such that $[e_i^*; l_i^*] \subseteq [e_i; l_i]$. Given a routing, we define the latest preferred delivery time z_i^* as the latest time a vehicle can serve at site i in order to be able to serve the successor visits of i within their preferred time-windows. The definition of z_i^* is similar to the definition of z_i.

$$z_i^* = \begin{cases} l_i^* & \forall i \in Depots \\ \min(l_i^*, z_{i^+}^* - c_{i,i^+} - d_i) & \forall i \in Customers \end{cases}$$

Given a routing R and service times $S = [s_i | i \in Sites]$, the violations of the soft time-windows constraint is defined as the number of customers who are not served within their soft time-windows: $ViolStw(R, S) = |\{i \in Customers \mid s_i \notin [e_i^*; z_i^*]\}|$. Please note that setting a service time $s_i > z_i^*$ will induce at least one unit of penalty for subsequent visits of i (and maybe more).

The VRPTW calls for a routing valid wrt the capacity and hard time-windows constraints, and minimizing the lexicographical objective $f(R, S) = \langle |R|, ViolStw, \sum_{i \in Sites} c_{i,i^+} \rangle$.

8.2 An Expressive Model

Let n be the number of clients to visit and K the number of available vehicles. We represent the depot by K dummy visits ranging from $n+1$ to $n+K$.

Variables A routing is represented as a collection of K variables $\mathcal{X} = [S_1, \ldots, S_K]$ representing sequences. The domain D of these variables is the set of possible sequences on the elements 1 to $n+K$. The number of elements in a sequence S is denoted $|S|$. For all i such that $1 \leq i < |S|$, we denote $S[i]$ the element at the i^{th} position in the sequence S.

Note that the values y_i, z_i, z_i^* are derived from the routing and these are not decision variables. We will also see that service times s_i can be derived frow the routing too.

The VRP with hard and soft time-windows to solve is the OCSP $\mathcal{P} = \langle f, \mathcal{C}_{VRP} + \mathcal{C}_2, \mathcal{X}, D \rangle$, where f is the objective function, $\mathcal{C}_{VRP}$ is the structural constraint (whether the variables represent a routing) and $\mathcal{C}_2$ is the hard constraints to be respected by all routings. These are described here below.

Structural Constraint In order for an assignment σ to represent a valid routing, we enforce that (a) the first visit of any vehicle k is the dummy visit $n+k$ representing the depot, and (b) the sequences represent a partition of the visits 1 to $n+K$.

The structural global constraint $\mathcal{C}_{VRP}$ is such that $\mathcal{C}_{VRP}(\sigma) = 0$ iff the conditions (a) and (b) are respected. A routing is an assignment σ such that $\mathcal{C}_{VRP}(\sigma) = 0$. We denote the position of element i in sequence S by S_i^{rank} and the subsequence of length m beginning with i by $S[i;m]$. A subsequence $S[i;m]$ is valid if $i \in S$ and $S_i^{rank} + m - 1 \leq |S|$.

Hard constraints The hard constraint $\mathcal{C}_2$ for the vehicle routing problem with hard/soft time-windows is $\mathcal{C}_2 = \mathcal{C}_{capa} + \mathcal{C}_{Htw}$ where $\mathcal{C}_{Htw}$ and $\mathcal{C}_{capa}$ are the hard time-windows and capacity constraint respectively. The violations of these constraints are defined as follows

$$\mathcal{C}_{Htw}(\sigma) = \sum_{i=1}^{n+K} \max(0, y_i - z_i) \tag{8.18}$$

$$\mathcal{C}_{capa}(\sigma) = \sum_{k=1}^{K} \max(0, \sum_{i \in S_k} d_i - Q) \tag{8.19}$$

Objectives The objective function for this problem is $f(\sigma) = \alpha.card(\sigma) + \beta.\mathcal{C}_{Stw}(\sigma) + dist(\sigma)$ with $\alpha, \beta > 0$ where $card$ is the function counting the number of vehicle used, $\mathcal{C}_{Stw}$ is the violation of the soft time-windows and $dist$ is the function giving the total distance of a routing.

We define

$$card(\sigma) = \Big|\{S_k \in \mathcal{X} \Big| \ |\sigma(S_k)| > 1\}\Big| \tag{8.20}$$

$$dist(\sigma) = \sum_{i=1}^{n+k} c_{i,i^+} \tag{8.21}$$

$$\mathcal{C}_{Stw}(\sigma) = \sum_{i=1}^{n+K} \Big(\max(0, s_i - l_i^*, e_i^* - s_i) > 0 \Big) \tag{8.22}$$

where s_i are the service time of customer i. In our approach, *we serve a client as soon as possible while trying to minimize the violations of the soft time-windows constraint*:

$$s_i = \max(y_i, s_{i^-} + d_{i^-} + d_{i^-,i}, \min(e_i^*, z_i^*)) \qquad \forall i \in Customers$$

Indeed, the soonest a customer i can be served is at time $\max(y_i, s_{i^-} + d_{i^-} + d_{i^-,i})$ (from Corollary 1). In order to minimize the violations of the

soft time-windows constraint, we would like to serve customer i at time $\min(e_i^*, z_i^*)$. Serving customer i at time e_i^* would induce no violation for customer i. However, if $z_i^* < e_i^*$, at least one subsequent customer of i will be served outside his preferred time-window. In this case it is desirable to serve i at time z_i^*; the preferred time-window of customer i will be violated, but we ensure that the preferred time-windows of all subsequent customers of i will be respected. Note that if $z_i^* < e_i^*$ it is impossible to achieve a zero-violation for the soft time-windows constraint.

Neighborhoods

We define four basic moves on sequences (Example 1). Let S be a sequence and $i, j \in S$. The move $reverse(S, i, j)$ reverses the subsequence from i to j in S. Let $S_1 \neq S_2$ be two sequences, $i \in S_1, j \in S_2$ and $S_1[i; m], S_2[j; n]$ be two valid subsequences. The moves are defined as follows: $insert(S_1, i, m, S_2, j)$ inserts the subsequence $S_1[i; m]$ in S_2 right after j, $remove(S_2, j, n)$ removes the subsequence $S_2[j, n]$ from S_2 and $replace(S_1, i, m, S_2, j, n, r)$ inserts $S_1[i; m]$ in S_2 at rank r and then removes $S_2[j; n]$ from S_2.

Example 1 We illustrate here the effect of the moves defined on sequences. Note that the sequence S_1 is not modified by any of these moves.

Input	Move	Result
$S_1 = [1, 2, 3, 4]$ $S_2 = [5, 6, 7, 8, 9]$	$reverse(S_2, 6, 8)$	$S_2 = [5, 8, 7, 6, 9]$
	$insert(S_1, 2, 2, S_2, 7)$	$S_2 = [5, 6, 7, 2, 3, 8, 9]$
	$remove(S_2, 7, 3)$	$S_2 = [5, 6]$
	$exchange(S_1, 1, 3, S_2, 6, 2, 5)$	$S_2 = [5, 8, 1, 2, 3, 9]$

We also define two moves that better improves the objective function f by inserting subsequences at a position minimizing the objective.

$$insert(S_1, i, m, S_2, f) \equiv insert(S_1, i, m, S_2, j)$$

where j is such that $\Delta_f(insert(S_1, i, m, S_2, j), \sigma)$ is minimum, and

$$exchange(S_1, i, m, S_2, j, n, f) \equiv exchange(S_1, i, m, S_2, j, n, r)$$

where r is such that $\Delta_f(exchange(S_1, i, m, S_2, j, n, r), \sigma)$ is minimum.

The moves presented here above must be encoded as a MoveGraph in order to use our generic cyclic search algorithm. A MoveGraph is a graph that is a special encoding of the moves: each edge corresponds to a move.

Definition 20. *We define the MoveGraph* $MG_{VRP}(f, L)(\sigma) = (V = V_1 \cup V_2, E, \eta)$ *where*

- $V_1 = \{(i,m) | 1 \leq i \leq n, \quad 1 \leq m \leq L, i \in Customers,$ $S[i;m]$ *is valid with* $i \in \sigma(S), S \in \mathcal{X}\}$
- $V_2 = \mathcal{X} = \{S_1, \ldots, S_K\}$
- $E = \{(a,b) \in V \times V | \quad a \in V_1 \vee b \in V_1$ $\wedge a, b$ *correspond to different routes*$\}$
- $\eta(a,b) =$

$$\begin{cases} exchange(S_k, i, m, S_{k'}, j, n, f) & \text{if } a = (i,m), b = (j,n) \in V_1 \\ & \text{with } i \in \sigma(S_k), j \in \sigma(S_{k'}) \\ insert(S_k, i, m, S, f) & \text{if } a = (i,m) \in V_1, b = S \in V_2 \\ & \text{with } i \in S_k, S \neq S_k \\ remove(S, i, m) & \text{if } a = S_k \in V_2, b = (i,m) \in V_1 \\ & \text{with } i \in \sigma(S) \end{cases}$$

The MoveGraph used for the VRPTW is described in Definition 20 and illustrated in Example 2. The nodes represents either valid subsequences $S_k[i;m]$ of the current routing, or entire routes $r^{(k)}$. The moves $insert, remove$ and $exchange$ are represented by edges. There is an edge between two nodes only if they correspond to different routes in order to maintain the consistency of the routing structure. A subsequence beginning with customer i correspond to the route $r^{(k)}$ such that $i \in r^{(k)}$.Note that for the moves $remove$, the semantic of the move $\eta(a,b)$ does not depend on a.

This MoveGraph is especially useful because any independant cycle corresponds to moves whose application respects the structural vehicle routing constraint, as stated by the following property.

Proposition 12. *The MoveGraph* $MG_{VRP}(f, L)(\sigma)$ *is cycle-consistent wrt the Vehicle Routing structure.*

Proof. The vehicle routing structure is respected iff (a) the first visit of any vehicle k is the dummy visit $n + k$ representing the depot, and (b) the sequences represent a partition of the visits 1 to $n + K$.

We must prove that if a assignment σ satisfies these conditions, then applying the moves of an independant cycle of $MG_{VRP}(f, L)(\sigma)$ will still respect (a) and (b)

(a) Only subsequences $S_k[i;m]$ with $i \notin Depots$ are represented in the MoveGraph. Thus the depot will never be moved.

(b) Given any subsequence $S_k[i;m]$, it is represented by a node $b = (i,m) \in V_1$, the only moves that may remove these visits from the current routing are *remove* and *exchange* moves that are represented in the MoveGraph by an ingoing edge (a,b). Reciprocally, the only moves inserting these visits into routes are *insert* and *exchange* moves represented by outgoing edges (b,c). Thus if a cycle C contains a move removing the subsequence $S_k[i;m]$, then it necessarily contains a move reinserting this same subsequence to another route. Reciprocally, if a cycle C contains a move inserting the subsequence $S_k[i;m]$ into a route, then it also contains a move removing this same subsequence from its current route.

So if a cycle contains only nodes representing disjoint subsequences, then by applying such cycle, there will be neither lost visits nor duplicated visits. As two moves are independant iff they modify different routes, the subsequences moved by applying an independant cycle will all belong to different routes and thus be disjoint.

□

The neighborhood $VLSN(\mathcal{P}, \sigma, MG_{VRP})$ contains an exponential number of candidates but our generic search procedure presented in Section 4.3 can search it heuristically in polinomial time. This emphasizes the usefulness of the MoveGraph $MG_{VRP}(f,L)(\sigma)$ as it completely defines a new VLSN.

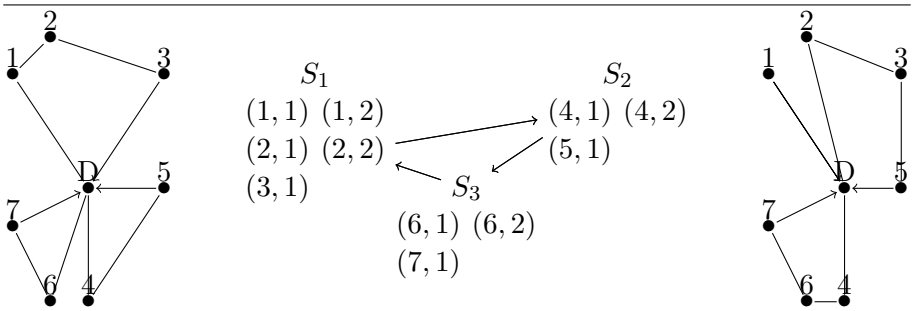
8.3 Capturing Human Knowledge in Search

Our VLSN search abstractions are open and flexible. They can be modified according to some expert's knowledge of the problem. We illustrate it here by mean of a multi-stage method to minimize the lexicographical objective $\langle card, \mathcal{C}_{Stw}, t \rangle$. Secondary objectives are used to better guide the search, and the search procedures are modified to better find improving cycles for the VRPTW.

Our multi-stage approach first minimizes the number of vehicles used, then the soft time-windows violations, and finally the overall distance.

We try to maintain the quality of the routes wrt the objective function as high as possible by searching for the best *reverse* move and

Example 2 Example of MoveGraph.



Let $n = 7$, $\sigma \in \Lambda$ such that $\sigma(S_1) = [8, 1, 2, 3], \sigma(S_2) = [9, 4, 5], \sigma(S_3) = [10, 6, 7]$. The routing described by σ is illustrated on the left. The MoveGraph $MG_{VRP}(dist, 2)(\sigma) = \langle V, E, \eta \rangle$ is illustrated in the middle, where *dist* is the function of the overall distance of a routing. The set of nodes V is entirely represented. By sake of clarity, only three edges are represented. The edge $((2, 2), (4, 1))$ corresponds to the move $exchange(S_1, 2, 2, S_2, 4, 1, t)$. Similarly $\eta(((4, 1), S_3)) = insert(S_2, 4, 1, S_3, t)$ and $\eta((S_3, (2, 2))) = remove(S_1, 2, 2)$. These moves are independant, so applying them respects the structural constraint $\mathcal{C}_{VRP}$. These moves are also compositional. The routing obtained is illustrated on the right. Note that visit 4 is inserted right before visit 6 because this position minimizes *dist*.

applying it before searching the cyclic VLSN at all these steps. This is repeated until there is no more *reverse* move improving the current solution wrt the current objective.

A) Reducing the number of vehicles used We select a vehicle and visits are moved one at the time to empty it. This process is repeated until no more vehicle can be emptied. However the hard time-windows often makes it impossible to move any visit. Similarly as in [BH04], we use two secondary objectives to guide the search towards solutions with enough space in the routes to allow the selected visit to be moved to another vehicle.

Given a vehicle k, we define the **minimal delay** of visit $i \in S_{k'}$ $(k \neq k')$ as the minimal violation of the hard time-windows constraint incurred by the insertion of visit i into vehicle k:

$$mdl_{k,i}(\sigma) = \min_{j \in S_k} \Delta_{\mathcal{C}_{Htw}}(insert(S_{k'}, i, 1, S_k, j), \sigma) \tag{8.23}$$

We also define the total minimum delay wrt visit i by $mdl_i(\sigma) =$

$\sum_{k=1}^{K} h(mdl_{k,i})$ where h is a function favoring small values: here we use $h(x) = \frac{C}{x} - \frac{x^2}{C'}$, where C, C' are two constants. Some experimental analysis showed that the values $C = 100000$ and $C' = 1000000$ worked well for the instances studied in our experimental section.

We also use the **squared cardinality** of visit i that favors solutions with less visits in the vehicle serving customer i:

$$exp_i(\sigma) = (ub_{visits} - |S_k|)^2 \qquad \text{with } i \in S_k \tag{8.24}$$

Where ub_{visits} is an upper bound on the number of customers visited by a vehicle. Here we set $ub_{visits} = 15$, that is function of the instances used in our experimental section. In order to reduce the number of vehicle used, we optimize the OCSP $\mathcal{P}_{card} = \langle\langle card, exp_i, mdl_i\rangle, \mathcal{C}_{VRP} + \mathcal{C}_2, \mathcal{X}, D\rangle$ where i is the visit we want to remove from its current vehicle. We iteratively apply the best negative candidate that is found in $VLSN_{cycle}(\mathcal{P}_{card}, \sigma, MG_{VRP}(f, 2), \{s\})$ with $s \in V$ selected randomly. If no cycle is found, the search procedure is repeated for different start nodes until a compositional and independant cycle is found or all start nodes have been considered. The exact search procedure is depicted through lines 1 to 12 in Algorithm 6. Note that because we want to empty a given vehicle S_k, we *freeze* this vehicle: the *insert* and *exchange* moves adding visits to S_k are not considered during the search for a cycle as they would increase the number of visits in S_k.

B) Reducing the soft time-windows violations In order to reduce the soft time-windows violations, we optimize the OCSP $\mathcal{P}_{card} = \langle f_2 = \langle card, \mathcal{C}_{Stw}\rangle, \mathcal{C}_{VRP} + \mathcal{C}_2, \mathcal{X}, D\rangle$ by iteratevely applying the first negative candidate found in $VLSN_{cycle}(\mathcal{P}_{card}, \sigma, MG_{VRP}(f_2, 2))$. We perform $nbIterStw$ iterations at this step. This search is depicted through lines 13 to 15 in Algorithm 6.

C) Reducing the overall distance The overall distance is minimized by optimizing the OCSP $\mathcal{P}_{card} = \langle f_3 = \langle card, 10.\mathcal{C}_{Stw} + t\rangle, \mathcal{C}_{VRP} + \mathcal{C}_2, \mathcal{X}, D\rangle$ by iteratevely applying the best negative candidate that is found in $VLSN_{cycle}(\mathcal{P}_{card}, \sigma, MG_{VRP}(f_3, 2), V')$ with a set $V' \subseteq V$ of 20 start nodes selected randomly. This search is depicted through lines 17 to 19 in Algorithm 6.

```
1  while the algorithm has not run for timeLimit seconds do
2      select ( k in 1..K ) do
3          freeze(S_k);
4          minimize ⟨card, t⟩ by applying the first cycle found in
           VLSN_cycle(P_card, σ, MG_VRP) while such cycle exists;
5          while |S_k| > 1 do
6              select ( i ∈ S_k with the smallest hard time windows )
               do
7                  while cycle found do
8                      for s ∈ V do
9                          minimize ⟨card, exp_i, mdl_i⟩ by applying the
                           best cycle in
                           VLSN_cycle(P_card, σ, MG_VRP, {s});
10                         if cycle found then break;
11                     if no cycle found ∀s ∈ V then break;
12         unFreeze(S_k);
13 while it < nbIterStw do
14     minimize f = ⟨card, Stw⟩ by applying the first cycle found;
15     if no cycle found then break;
16 restore the best solution found in the loop 13-15;
17 while true do
18     minimize ⟨card, 10.Stw + dist⟩ by applying the best cycle in
       VLSN_cycle(P_card, σ, MG_VRP, V′);
19     if no cycle found then break;
```

8.4 A Simple and Efficient Implementation

The VRPTW illustrates that our framework is open to new variables and moves. New computational domains can thus be integrated into our framework. Indeed the VRPTW cannot be totally modelled as a partitionning or a permutation problem. Thus the differentiable invariant presented so far cannot be reused. In order to use our generic VLSN search procedures, we have to implement (1) new variables to represent a routing, (2) the moves modifying these variables, (3) the differential invariants to be used in the model, and (4) the MoveGraph fully describing the VLSN. The VRPTW is thus a worst-case problem in terms of implementation requirements. However these requirements can still be developped easily. We detail the integration of vehicle routing problems in our framework here below.

Routing and Sequences

Sequences are important for modeling the VRPTW. Indeed we represent a vehicle routing by a collection of K sequences (Listing 8.1). A sequence is encoded as an array of successors `succ` and the first element of this sequence (these fully describe a sequence). For example `succ[2]` is the next element after element 2 and the variable `first` represents the first element of the sequence. The array of the predecessors and the set of values of a sequence are also maintained from the successor array.

We represent a routing by a collection of K sequences over the elements 1 to $n + K$ (Listing 8.2). Each sequence represents the route of one vehicle. The visits from 1 to n represent clients to be visited and the depot is represented by K dummy visits numbered from $n + 1$ to $n + K$. In order to be space-efficient, the array of successors is shared among the K sequences. This is well-defined as a given visit can only be served by a unique vehicle and has thus a unique successor. The first visit of the k^{th} sequence is constantly the visit $n + k$ representing the depot.

Moves on Sequences

The class `Sequence`(Listing 8.1) also contains the definition of the four basic moves defined on sequences: insert, remove, exchange and reverse. For each move, two methods are provided. The first applies the move and the second records the corresponding output variables. The complete implementation of the move `reverse` is given as an example.

```
class Sequence{
```

```
  var{int}[] succ;
  var{int}   first;
  Sequence(Solver<VLSN> m, int n);
  int getSize();
  var{set{int}} getValues();
  var{int} getFirst();
  var{int} getSucc(int i);
  var{int} getPred(int i);
  var{int} getRank(int i);

  void setInitialSolution(int[] seq);
  void reverse(int i, int h){
    succ[i] := succ[h];
    pred[(int)succ[h]] := i;
    pred[h] := pred[i];
    succ[(int)pred[i]] := h;
    int c = i;
    while ( c != h ) {
      pred[c] := (int)succ[c];
      succ[(int)succ[c]] := c;
      c = succ[c];
    }
  }
  void registerReverse(int i, int h, VLSNTracker t){
    t.write(this);
    t.write(succ[i]) ;
    t.write(pred[(int)succ[h]]) ;
    t.write(pred[h]) ;
    t.write(succ[(int)pred[i]]) ;
    int c = succ[i];
    while ( c != h ) {
      t.write(pred[c]) ;
      t.write(succ[(int)succ[c]]) ;
      c = succ[c];
    }
    int rh = rank[h];
    int ri = rank[i];
  }
  void exchange(int i, int m, Sequence s,int j, int n,
  int r);
  void registerExchange(int i, int m, Sequence s,int j,
  int n, int r, VLSNTracker t);
  void insert(int i, int m, Sequence s,int j);
  void registerInsert(int i, int m, Sequence s,int j,
  VLSNTracker t);
  void remove(int i, int m);
  void registerRemove(int i, int m, VLSNTracker t);
}
```

Listing 8.1: API of sequences

```
class Routing{
  Sequence[] seqs;
  var{int}[] p;     // p[i] represents the index of
                    //  the sequence containing i
  Routing(Solver<VLSN> m, int n, int K){
    seqs = new Sequence[k in 1..K](n);
  }
  Sequence getSequence(int k){ return seqs[k];}
  Sequence getSequenceOf(int i){ return seqs[p[i]];}
  void insert(int i, int l, int j);
  void remove(int i, int l);
  void exchange(int i, int l, int j, int k, int r);
}
```

Listing 8.2: Routing object using Sequences.

The Model and Differential Invariants

The COMET model for the Vehicle Routing Problem reflects the model of Section 8.2 (Listing 8.3). It describes the COP $\mathcal{P}_{VRPTW}$ and contains three parts: (1) the description of the variables (and domains), (2) the definition of the objective function, and (3) the constraints .

```
Solver<VLSN> m();
Sequence Seqs[1..K](m,n+K);

var{int} coefMDL(m);
var{int} coefStw(m);
var{int} coefSq(m);
var{int} coefDist(m);
var{int} coefP[1..K](m);

var{int} visitToMove(m);

ConstraintSystem<Sequence> S(m);
S.post(CapacityConstraint(m, Seqs, demands,Q));       //
 Capacity constraint
S.post(HardTW(Seqs, distances, duration, earliest,
latest));
S.close();

SumFunction<Sequence> O(m);
O.post(10000000*Card(Seqs));
O.post(coefSq*sum(i in 1..K) (coefP[i]*(15-Card(Seqs[i])
)^2));
O.post(coefDist*sum(i in 1..K) SeqDistanceSym(m, Seqs[i
], costs));
```

```
O.post(coefStw*STWSeq(Seqs, costs, service, earlydate,
duedate, Wmax));
O.post(coefMDL*MinimalDelayInvariant (visitToMove, Seqs,
  Htw, n));
O.close();

m.close();
```

Listing 8.3: Model for the VRPTW.

We already have described the implementation of the `Sequence` variables and their domains. The coefficients allow to implement the different lexicographical objectives presented in Section 8.3. The integer variable `visitToMove` represents the visit to be removed when reducing the number of vehicles of the routing. It is used for the minimal delay objective.

The constraints and objective functions are described by differential invariants (Listing 8.4). For each of the moves, there are two methods defined: one differentiating the invariant wrt this move, and a second returning the input variables of this move. The implementation of the `getDelta` methods is straightforward for all objectives and constraints used in the model from their definition. For each of these moves, the differentiation only depends on the sequence modified by the move, so the `getVariables` method only add this sequence as an input variable in the `VariablesCollector`.

```
interface SequenceDifferentialInvariant{
  var{int} value();
  set{int} registerVariables();
  int getInsertDelta(Sequence si, int i, int m, Sequence
   sj, int j);
  void getInsertVariables(Sequence si, int i, int m,
  Sequence sj, int j,
    VariablesCollector t);
  int getExchangeDelta(Sequence si,int i,int m,Sequence
  sj, int j, int n, int r);
  void getExchangeVariables(Sequence si, int i, int m,
  Sequence sj, int j, int n,
    int r, VariablesCollector t);
  int getRemoveDelta(Sequence sj, int j, int n);
  void getRemoveVariables(Sequence sj, int j, int n,
  VariablesCollector t);
  int getReverseDelta(Sequence S, int i, int j);
  void getReverseVariables(Sequence S, int i, int j,
  VariablesCollector t);
}
```

Listing 8.4: Interface for the differential invariants to be used with the `Routing` object.

The Search and MoveGraph

Once a MoveGraph is implemented, our generic search algorithms can be used to search VLSN. Because we can reuse the complex shortest-cycle algorithm, only the MoveGraph MG_{VRP} must be implemented to completely describe the search. The MoveGraph is described through the implementation of the interface depicted in Listing 5.8. There is no difficulty in translating the MoveGraph from Definition 20 into COMET (Listing 8.5).

```
class ExchangeVRPVLSN extends MoveGraph{
  SequenceConstraint C;
  SequenceFunction O;

  VRP v;
  int L;
  range nodes;

  int[] a;
  int[] b;

  ExchangeVRPVLSN(Solver<VLSN> vs, SequenceConstraint C,
   SequenceFunction O, VRP v, int L){
    nodes = 1..L*v.getN()+v.getK();
    a = new int[i in nodes] = (i-1)%v.getN()+1;
    b = new int[i in nodes] = (i-1)/v.getN()+1;
  }
  range getNodes(){ return nodes; }
  bool isValid(int i){
    if ( b[i] == 0) return true;
    Sequence S = (b[i] == 0) ? v.getSequence(a[i]) : v.
    getSequenceOf(a[i]);
    return S.getRank(a[i]) + b[i] <= S.getSize();
  }
  bool isMove(int i, int j){
    Sequence Si = (nn[oi,2] == 0) ? v.getSequence(i) : v
    .getSequenceOf(i);
    Sequence Sj = (nn[oj,2] == 0) ? v.getSequence(j) : v
    .getSequenceOf(j);

    if (Si.getId() == Sj.getId()) return false;
    if (li+lj <= 0 && li+lj != -2) return false;
    return true;
```

```
  }
  void applyMove(int oi, int oj){
    Sequence Si = (b[i] == 0 ) ? v.getSequence(a[i])
                               : v.getSequenceOf(a[i]);
    Sequence Sj = (b[j] == 0 ) ? v.getSequence(a[j])
                               : v.getSequenceOf(a[j]);

    if ( b[j] == 0 ) {         //Insert
      selectMin(jp in Sj.getValues() :
                C.isInsertFeasible(Si, a[i], b[i],Sj,
                 jp),
                d = O.getInsertDelta(Si, a[i], b[i],
                Sj, jp)
              )(d){
        v.insert(a[i], b[i],jp);
      }
    }else if ( b[i] == 0 ) {    // Remove
      if (! C.isRemoveFeasible(Sj, a[j], b[j]) ) return;
      else v.remove(a[j], b[j]);
    }else {                      // Exchange
      selectMin(r in 1..Sj.getSize() :
                (r <= Sj.getRank(a[j]) || r > Sj.
                getRank(a[j])+b[j])
                && C.isExchangeFeasible (Si, a[i], b[i
                ],Sj, a[j], b[j],r),
                d = O.getExchangeDelta(Si, a[i], b[i],
                Sj, a[j], b[j],r)
              )(d){
        v.exchange(a[i],b[i],a[j],b[j],r);
      }
    }
  }
  bool isMoveFeasible(int i, int j);
  int getMoveDelta(int i, int j);
}
```

Listing 8.5: MoveGraph for the VRPTW. The parameter `L` is the maximum length of the subroutes to be exchanged during the VLSN search. A node $i|0 < i \leq nL$ represents the subroute beginning at visit $a[i]$ and containing $b[i]$ consecutive visits. A node $i > nL$ represents the entire route $i - nL$. The method `isValid(i)` checks whether the subroute represented by the node i is valid.

Once the MoveGraph is defined, our generic search algorithm can search for compositionnal and independant cycles (Section 5.6).

8.5 State-of-the-Art Solutions on Well-known Benchmarks

This section assesses the efficiency of our CBVLSN approach to solve the Vehicle Routing Problem with Soft Time-Windows.

Benchmarks

This section compares our CBVLSN approach to other solution methods on several well-known benchmarks. These benchmarks were first proposed by [Bal93] and are variations of the Solomon instances for the VRP with hard time-windows. For these instances, a time-window $[a_i, b_i]$ is specified for each customer i. The depot is also constrained by the time-window $[a_0, b_0]$.

Different definitions of the hard and soft time-windows have been investigated in the literature. They were classified into six *types* in [FEL08]. Our approach using hard and soft time-windows can deal with all these different definitions. In this section, we focus on type 1 and type 3 because state-of-the-art approaches mainly consider these two types.

Type 1 In [TBG+97], a vehicle can arrive earlier at a customer i, but cannot serve him earlier than his earliest service time ($e_i = e_i^* = a_i$). However, the vehicle can serve the client later, with a penalty linear in the delay ($l_i^* = b_i$). For the customers, there is no hard latest service time ($l_i = \infty$), but for the depot, the time-window $[a_0, b_0]$ is hard. To our knowledge, these type of problem has been investigated in [TBG+97, FEL08, Fig09].

Type 3 In [Bal93], the time-window is considered as soft ($e_i^* = a_i$ and $l_i^* = b_i$). There is also a hard time-window for each customer i, requiring that each customer has to be served within a certain percentage P of the total route duration $D = b_0 - a_0$:

$$\begin{aligned} e_i &= a_i - D\frac{P}{100} \\ l_i &= b_i + D\frac{P}{100} \end{aligned}$$

Moreover, a vehicle is allowed to arrive earlier than e_i at a customer i but cannot wait more than w_{max} before serving the customer. The parameter w_{max} is also expressed as a percentage of the total route duration D: $w_{max} = D\frac{W}{100}$.

Typical values of P and W are $0, 5$ or 10. To our knowledge, these type of problem has been investigated in [Bal93, CR04, Fig09, FEL08].

Experimental setup

The experiments were run on Intel Q6600 2,4GHz. Our algorithms were not parallelized (a single core was used). We used a time limit of 250 seconds to reduce the number of vehicles and performed 200 iterations to reduce the violations of the soft time-windows constraint.

Experimental Results

The results achieved with our COMET implementation of the CBVLSN framework for solving the VRPSTW(type 1) are depicted in Table 8.1. They are compared to the best published solutions among [TBG$^+$97, FEL08, Fig09]. Our approach found better solutions for all the 20 instances and managed to decrease the number of vehicles for 9 instances.

For Type 3, our method found better solutions for 14 out of 16 instances and decreased the number of vehicles used for 4 instances.

These results shows that our algorithms obtained state-of-the-art solutions for the Vehicle Routing Problem with soft Time-Windows. Our algorithm often finds routing using less vehicles than previously published solutions. Reducing the number of vehicles used is very important in real-life as it decreases significantly the operational cost of the routing.

	Best Published			CBVLSN			
Instance	*card*	$\mathcal{C}_{Stw}$	*dist*	*card*	$\mathcal{C}_{Stw}$	*dist*	Time(s)
R101	12	44	1129	**11**	45	1318	499
R102	11	54	1059	**10**	38	1232	523
R103	10	66	1027	**9**	30	1032	599
R104	9	82	947	9	**12**	946	624
R105	11	58	1074	**10**	32	1181	573
R106	10	67	1047	10	**24**	1090	640
R107	10	76	988	**9**	20	985	555
R108	9	86	947	9	**8**	940	696
R109	10	72	1001	10	**20**	1118	629
R110	9	71	1013	9	**24**	1028	527
R111	10	74	983	**9**	19	1003	635
R112	9	83	941	9	**10**	1025	644
RC101	11	56	1255	11	**31**	1459	496
RC102	10	68	1230	10	**28**	1323	473
RC103	10	75	1155	10	**14**	1259	567
RC104	10	88	1084	**9**	12	1129	638
RC105	11	62	1220	**10**	31	1333	501
RC106	10	73	1150	10	**18**	1231	493
RC107	10	72	1123	10	**15**	1180	609
RC108	10	90	1072	**9**	14	1126	620
Average	55%	0%		100%	100%		

Table 8.1: Results for the Type 1. Our algorithm found the best solution for all the instances. It was able to improve the number of vehicle used for 45% of the instances. The numbers in bold-font indicate the component of the lexicographical objective that was improved and that lead to the best solution.

p_{max}	Instance	Best Published			CBVLSN			
		card	$\mathcal{C}_{Stw}$	*dist*	*card*	$\mathcal{C}_{Stw}$	*dist*	*time*
5%	R101	14	32	1633	**13**	58	1582	526
	R102	12	37	1404	12	**31**	1430	610
	R103	11	7	1374	**10**	32	1274	529
	R109	11	7	1393	**10**	38	1242	521
	RC101	13	7	1778	**12**	51	1696	425
	RC102	11	42	1375	11	**28**	1550	451
	RC103	10	17	1256	10	**15**	1362	511
	RC106	11	19	1336	11	**18**	1492	424
10%	R101	12	69	1376	12	**59**	1421	495
	R102	10	67	1173	10	**62**	1259	554
	R103	10	**24**	1274	10	26	1228	559
	R109	10	47	1116	10	**28**	1200	522
	RC101	11	57	1322	11	**51**	1491	466
	RC102	11	26	1367	11	**23**	1341	489
	RC103	10	**15**	1228	10	24	1344	464
	RC106	10	51	1160	10	**33**	1327	540
		75%	12.5%		100%	87.5%		

Table 8.2: Results for the Type 3. Our algorithm found the best solution for 87.8% of the instances and was able to reduce the number of vehicles used for 25% of them. Note that for the two instances not improved by our algorithm, the solution found is close to the best known solution. The numbers in bold-font indicate the component of the lexicographical objective that was improved and that lead to the best solution. Time is expressed in seconds

Chapter 9

Conclusions and Future Works

The objective of this research was the merge of two different technologies: Constraint-Based Local Search and Very Large-Scale Neighborhood. The advantages of these two technologies broaden the field of application of each other. Now VLSN techniques profit from the high-level modelling of CBLS. First, incremental algorithms searching a VLSN are encapsulated inside expressive objects that can be reused to solve many different applications. Second, the inherent modularity of CBLS enables to build VLSN search algorithms more quickly, while preserving the efficiency of dedicated algorithms. Finally, the high-level of reasoning of CBLS allows to implement more complex VLSN approaches whose development time would be prohibitive without this framework.

On the other side, CBLS also profits from the integration of VLSN. First, CBLS approaches can now rely on exponential-sized neighborhoods. This broadens the range of applications that can be solved by CBLS. Second, VLSN as compounding moves can be considered as a meta-heuristic; from a local search approach performing only one single move per iteration, one can design a VLSN approach that performs many of them per iteration. Our approach significantly helps in designing a VLSN search algorithm from existing CBLS algorithms. Third, new original solutions can also be developped mixing the existing concepts of CBLS with the concepts designed in this work, opening new research perspectives for CBLS.

First we have highlighted that VLSN are efficient because they consider atomic moves that violates some constraints. Such moves would not be considered by standard Local Search approaches. This larger set of considered moves allows VLSN search algorithms to escape local optima

of standard Local Search approaches. Second we expressed VLSN search algorithms as three separated high-level concepts: a model, a description of the VLSN structure and the search. The model captures the input and output variables that are required to compute which moves can be applied at the same iteration. The MoveGraph captures the structure of the VLSN that is independent from the model and the search algorithm. The search captures the efficient algorithms that selects the best candidate in a VLSN. Now we can synthetize state-of-the-art VLSN search algorithms by using these modular components that can be defined independently.

From a technical point of vue, the contributions of this thesis are the following:

1. We identified that a large part of the definition of the improvement graph, used in existing works, is independent of the problem to solve. This part was captured in the concept of **MoveGraph**.

2. We showed that VLSN approaches are beneficial because they can consider atomic moves that violate a selected constraint. The selected constraint is maintained by appropriately selecting the set of moves to apply at each iteration. This is captured by the concept of **cycle consistency**.

3. We raised that independence does not completely specify which moves can be selected together in a VLSN search algorithm. We introduced the concept of **compositionality** to complete the criteria that a set of moves has to respect in order to be applied together.

4. We enlightened that compositionality enables to specify which edges have to be updated in the improvement graph after several moves have been applied.

5. We demonstrated that **input and output variables** enable the automatic computation of the variable-independence and variable-compositionality. These notions are stricter than theoretical independence and compositionality, but equivalent for most problems in practice; this is the case for all problems considered in this thesis. Input and output variables are natural to express in function of the constraint or the objective function.

6. We illustrated that VLSN search algorithms implemented using our CBVLSN abstractions performs exactly the same operations than a dedicated VLSN search algorithm, and is comparable in time with this same dedicated implementation.

7. We highlighted that checking compositionality enhances the efficiency of VLSN search algorithms compared to the same algorithm only checking independence.

8. We found that our high-level abstractions allow the design of a VLSN search algorithm that solves a problem with complex constraints and objective functions.

These technical contributions have the following respective significance:

1. We can define several VLSN generically and use them for solving several different problems

2. It is well understood that the more a constraint prunes the set of neighbors, the more a VLSN designed wrt this constraint will be efficient.

3. There is a well-defined criteria to ensure that the differentiation of a set of moves can be computed accurately and efficiently.

4. A necessary condition determines if the differentiation of a move has changed from the last iteration.

5. Checking whether a set of moves is independent and compositional wrt a problem can be automated, based on a natural extension of the CBLS API that provides the input and output variables.

6. We know that all the benefits of our approach in terms of modularity, flexibility and easiness have no slow down counterparts compared to dedicated algorithms.

7. Compositionality has to be considered in VLSN search algorithms.

8. Our modular components can be reused to quickly assess the efficiency of a VLSN approach, and the VLSN search can be enhanced by the transparent addition of heuristics.

There are several perspectives opened by this research. First, VLSN are powerful because they consider moves that violate a specific constraint and select several of them such that the set of applied move do not violate it. In the literature, VLSN were designed only wrt the permutation and partitioning constraints. A perspective of research is the design of VLSN focusing on other constraints. This would broaden the range of problems that can be solved by VLSN approaches.

Second, VLSN and Large Neighborhood Search (LNS) may have interesting properties if used bibtogether. Large Neigborhood Search departs from an initial solution. Then it unassigns several variables and uses Constraint Programming (CP) techniques to reassign them in order to improve the objective function. The choice of variables to relax is the critical aspect when designing a LNS algorithm. If the variables are *independent*, the CP algorithm will likely reassign them the values they had at the previous iteration. It is thus important to select *correlated variables* according to the computation of the objective function and the constraints. The input and output variables introduced in this thesis specify which variables are *independent*. The study of a generic LNS algorithm that selects the variables to relax based on the input and output variables may lead to interesting results.

Third, LNS and VLSN algorithms are likely to complement each other very well when solving complex problems. Indeed, VLSN algorithms modify several *independent* variables at the same iteration. At the opposite, LNS algorithms modify several *correlated* variables at the same iteration. Thus we believe that these approaches are likely to escape local optima of each other.

Fourth, this work showed that the input and output variables are useful to compute whether a set of moves can be differentiated efficiently and to incrementally build the improvement graph. They could also be useful for generic LNS variables selection as explained here above. Other usage of these variables may exist.

Fifth, the efficiency of CBVLSN search algorithms depends whether many moves have non-intersecting input and output variables. Thus how the problem is modelled impacts on the behaviour of the generic VLSN algorithms. Using low-level variables (such as boolean variables) will more likely lead to non-intersecting input and output variables. However the trend of Constraint-Based Local Search is to use high-level variables. Using a lower-level model for the input and output variables could have advantages when solving some applications.

Bibliography

[AAB+07] Salwani Abdullah, Samad Ahmadi, Edmund Burke, Moshe Dror, and Barry McCollum, *A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem*, Journal of the Operational Research Society **58** (2007), 1494–1502.

[AABD04] Salwani Abdullah, Samad Ahmadi, Edmund Burke, and Moshe Dror, *Applying ahuja-orlin's large neighborhood for constructing examination timetabling solution*, Proceedings of the Fifth International Conference on the Practice and Theory of Automated Timetabling, Lecture Notes in Computer Science, no. 3616, Springer, 2004, pp. 413–420.

[AABD07] ———, *Investigating ahuja-orlin's large neighbourhood search approach for examination timetabling*, OR Spectrum **29** (2007), 351–372.

[AC05] Ravindra K. Ahuja and Claudio B. Cunha, *Very large-scale neighborhood search for the k -constraint multiple knapsack problem*, Journal of Heuristics **11** (2005), 465–481.

[ADSV06] Avella, D'Auria, Salerno, and Vasil'ev, *A computational study of local search algorithms for italian high-school timetabling*, Journal of Heuristics (2006), 543–556.

[AGM+01] Ravindra K. Ahuja, Jon Goodstein, Amit Mukherjee, James B. Orlin, and Dushyant Sharma, *A very large-scale neighborhood search algorithm for the combined through and fleet assignment model*, Tech. Report 4388-01, Mit Sloan School of Management, dec 2001.

[AJOS02] Ravindra K. Ahuja, Krishna C. Jha, James B. Orlin, and Dushyant Sharma, *Very large-scale neighborhood search for*

the quadratic assignment problem, Tech. Report 4386-02, MIT Sloan School of Management, 2002.

[AKJO03] Ravindra K. Ahuja, Arvind Kumar, Krishna Jha, and James B. Orlin, *Exact and heuristic methods for the weapon target assignment problem*, Tech. Report 4464-03, MIT Sloan School of Management, jul 2003.

[ALO+02] Ravindra K. Ahuja, Jian Liu, James B. Orlin, Dushyant Sharma, and Larry A. Shughart, *Solving real-life locomotive scheduling problems*, Tech. Report 4389-02, MIT Sloan School of Management, april 2002.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, *Network flows: Theory, algorithms, and applications*, united states ed ed., Prentice Hall, February 1993.

[AOEOP02] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen, *A survey of very large-scale neighborhood search techniques*, Discrete Appl. Math. **123** (2002), no. 1-3, 75–102.

[AOP+02] R.K. Ahuja, J.B. Orlin, S. Pallottino, M.P. Scaparra, and M.G. Scutella, *A multi-exchange heuristic for the single source capacitated facility location problem*, Tech. Report 4387-02, MIT Sloan School of Management, 2002.

[AOS01] Ravindra K. Ahuja, James B. Orlin, and Dushyant Sharma, *Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem*, Mathematical Programming **91** (2001), 71–97.

[AOS03] ———, *A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem*, Operations Research Letters **31** (2003), 185–194.

[Bal93] Nagraj Balakrishnan, *Simple heuristics for the vehicle routeing problem with soft time windows*, The Journal of the Operational Research Society **44** (1993), no. 3, 279–287.

[BD95] Burkard and Deineko, *Polynomially solvable cases of the traveling salesman problem and a new exponential neighborhood*, Computing **54** (1995), 191–211.

[BDW98] Rainer E Burkard, Vladimir G Deineko, and Gerhard J Woeginger, *The travelling salesman and the pq-tree*, Mathematics of Operations Research **23** (1998), 613–623.

[Ben10] Thierry Benoist, *Characterization and automation of matching-based neigborhood*, CPAIOR'10, 2010.

[BH04] Russell Bent and Pascal Van Hentenryck, *A two-stage hybrid local search for the vehicle routing problem with time windows*, Transportation Science **38** (2004), 515–530.

[BH06] _____, *A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows*, Computers & Operations Research **33** (2006), 875–893.

[BO05] Bompadre and Orlin, *Using grammars to generate very large scale neighborhoods for the traveling salesman problem and other sequencing problems*, Integer Programming and Combinatorial Optimization **3509/2005** (2005), 437–451.

[BS01] E. Balas and N. Simonetti, *Linear time dynamic programming algorithms for new classes of restricted tsps: A computational study*, 2001, pp. 56–75.

[CDS08] Marco Chiarandini, Irina Dumitrescu, and Thomas Stützle, *Very Large-Scale neighborhood search: Overview and case studies on coloring problems*, Hybrid Metaheuristics (Christian Blum, Maria José Blesa Aguilera, Andrea Roli, and Michael Sampels, eds.), vol. 114, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 117–150.

[CFS+04] J. Carpenter, P.H.S. Funk, A. Srinivasan, J. Anderson, and S. Baruah, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, 2004.

[CL96] M. Carter and G. Laporte, *Recent developments in practical examination timetabling*, Practice and Theory of Automated Timetabling (1996), 1–21.

[CNP83] G. Cornuéjols, D. Naddef, and W. Pulleyblank, *Halin graphs and the travelling salesman problem*, Mathematical Programming **26** (1983), 287–294.

[Con00] Richard K Congram, *Polynomially searchable exponential neighbourhoods for sequencing problems in combinato-*

rial optimisation, http://eprints.soton.ac.uk/50630/, April 2000.

[CPvdV02] Richard K. Congram, Chris N. Potts, and Steef L. van de Velde, *An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem.*, INFORMS Journal on Computing **14** (2002), 52.

[CR04] W.C. Chiang and R. A. Russell, *A metaheuristic for the Vehicle-Routing problem with soft time windows*, The Journal of the Operational Research Society **55** (2004), no. 12, 1298–1310.

[CV90] J. Carlier and P. Villon, *A new heuristic for the traveling salesman problem*, RAIRO Operational Research **24** (1990), 245–253.

[DL86] Moshe Dror and Larry Levy, *A vehicle routing improvement algorithm comparison of a greedy and a matching implementation for inventory routing*, Comput. Oper. Res. **13** (1986), 33–45.

[Dus02] Sharma Dushyant, *Cyclic exchange and related neighborhood structures for combinatorial optimization problems*, Ph.D. thesis, Sloan School of Management - MIT, 2002.

[DW00] Vladimir G Deineko and Gerhard J Woeginger, *A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem*, Mathematical Programming **87** (2000), 519–542.

[EGN06] Bertrand Estellon, Frédéric Gardi, and Karim Nouioua, *Large neighborhood improvements for solving car sequencing problems*, RAIRO Operational Research Research **40** (2006), 355–379.

[EO06a] Ozlem Ergun and James B. Orlin, *A dynamic programming methodology in very large scale neighborhood search applied to the traveling salesman problem*, Discrete Optimization **3** (2006), 78–85.

[EO06b] ———, *Fast neighborhood search for the single machine total weighted tardiness problem*, Operations Research Letters **34** (2006), no. 1, 41–45.

[EOSF02] Ozlem Ergun, James B. Orlin, and Abran Steele-Feldman, *Creating very large scale neighborhoods out of smaller ones by compounding moves: A study on the vehicle routing problem*, Tech. Report 4393-02, MIT Sloan School of Management, oct 2002.

[EW66] L. R. Esau and K. C. Williams, *On teleprocessing system design: part II a method for approximating the optimal network*, IBM Systems Journal **5** (1966), no. 3, 142–147.

[FEL08] Zhuo Fu, Richard Eglese, and Leon Li, *A unified tabu search algorithm for vehicle routing problems with soft time windows*, Journal of the Operational Research Society **59** (2008), 663–673.

[FFT05] Roberto De Franceschi, Matteo Fischetti, and Paolo Toth, *A new ILP-based refinement heuristic for vehicle routing problems*, Mathematical Programming **105** (2005), no. 2-3, 471–499.

[Fig09] Miguel Andres Figliozzi, *An iterative route construction and improvement algorithm for the vehicle routing problem with soft time windows*, Transportation Research Part C: Emerging Technologies (2009), 668–679.

[FNS00] Antonio Frangioni, Emiliano Necciari, and Maria Grazia Scutella', *A multi-exchange neighborhood for minimum makespan machine scheduling problems*, Tech. report, Università di Pisa - Dipartimento di Informatica, 2000.

[FW90] R. Fahrion and M. Wrede, *On a principle of chain-exchange for vehicle-routeing problems (1-vrp)*, The Journal of the Operational Research Society **41** (1990), 821–827.

[GCT04] A. Grosso, F. Della Croce, and R. Tadei, *An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem*, Operations Research Letters **32** (2004), 68–72.

[GG05] Gutin and Glover, *Further extension of the tsp assign neighborhood*, Journal of Heuristics **11** (2005), 501–505.

[GGPS06] Michel Gendreau, Francois Guertin, Jean-Yves Potvin, and Rene Seguin, *Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries*,

Transportation Research Part C: Emerging Technologies **14** (2006), 157–174.

[GGYZ01] Fred Glover, Gregory Gutin, Anders Yeo, and Alexey Zverovich, *Construction heuristics for the asymmetric tsp*, European Journal of Operational Research **129** (2001), 555–568.

[GP97] F. Glover and A.P. Punnen, *The travelling salesman problem: new solvable cases and linkages with the development of approximation algorithms*, Journal of the Operational Research Society **48** (1997), no. 5, 502–510.

[GP02] Gregory Gutin and Abraham P. Punnen, *The traveling salesman problem and its variations*, Springer, 2002.

[GPB05] Celia A Glass and Adam Prügel-Bennett, *A polynomially searchable exponential neighbourhood for graph colouring*, March 2005.

[GR06] Glover and Rego, *Ejection chain and filter-and-fan methods in combinatorial optimization*, 4OR: A Quarterly Journal of Operations Research **4** (2006), 263–296.

[Gut99] Gregory Gutin, *Exponential neighborhood local search for the traveling salesman problem*, Computers and Operational Research (1999), no. 26, 313–320.

[GY99] Gregory Gutin and Anders Yeo, *Small diameter neighbourhood graphs for the traveling salesman problem: at most four moves from tour to tour*, Computers & Operations Research **26** (1999), 321–327.

[GYZ04] Gregory Gutin, Anders Yeo, and Alexei Zverovitch, *Exponential neighborhoods and domination analysis for the tsp*, ch. The Traveling Salesman Problem and Its Variations, Springer US, 2004.

[HK61] Michael Held and Richard M. Karp, *A dynamic programming approach to sequencing problems*, Proceedings of the 1961 16th ACM national meeting, ACM, 1961, pp. 71.201–71.204.

[HM03] Pierre Hansen and Nenad Mladenović, *Variable neighborhood search*, Handbook of Metaheuristics (Fred Glover and

Gary A. Kochenberger, eds.), vol. 57, Kluwer Academic Publishers, Boston, 2003, pp. 145–184.

[HM05a] Pascal Van Hentenryck and Laurent Michel, *Constraint-based local search*, MIT Press, October 2005.

[HM05b] _____, *Control abstractions for local search*, Constraints **10** (2005), no. 2, 137–157.

[HM06] Pascal Hentenryck and Laurent Michel, *Differentiable invariants*, Principles and Practice of Constraint Programming - CP 2006 (Frédéric Benhamou, ed.), vol. 4204, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 604–619.

[Hur99] Hurink, *An exponential neighborhood for a one-machine batching problem*, OR Spectrum **21** (1999), 461–476.

[IIK+05] T. Ibaraki, S. Imahori, M. Kubo, T. Masuda, T. Uno, and M. Yagiura, *Effective local search algorithms for routing and scheduling problems with general time-window constraints*, Transportation Science **39** (2005), 206–232.

[Kar79] Richard M. Karp, *A patching algorithm for the nonsymmetric traveling-salesman problem*, SIAM Journal on Computing **8** (1979), 561–573.

[LK73] S. Lin and B. W. Kernighan, *An effective heuristic algorithm for the travelling-salesman problem.*, Operations Research **21** (1973), 498.

[MBHS03] L. Merlot, N. Boland, B. Hughes, and P. Stuckey, *A hybrid algorithm for the examination timetabling problem*, Practice and Theory of Automated Timetabling IV (2003), 207–231.

[MO07] Carol Meyers and James B. Orlin, *Very Large-Scale neighborhood search techniques in timetabling problems*, Practice and Theory of Automated Timetabling VI (Edmund K. Burke and Hana Rudová, eds.), vol. 3867, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 24–39.

[MPS02] Michael J. Magazine, George G. Polak, and Dushyant Sharma, *A multi-exchange neighborhood search heuristic for an integrated clustering and machine setup model for*

pcb manufacturing., Journal of Electronics Manufacturing **11** (2002), 107.

[ÖKNP08] T. Öncan, S. Kabadi, K. Nair, and A. Punnen, *VLSN search algorithms for partitioning problems using matching neighbourhoods*, The Journal of the Operational Research Society **59** (2008), no. 3, 388.

[PK02] Abraham Punnen and Santosh Kabadi, *Domination analysis of some heuristics for the traveling salesman problem*, Discrete Applied Mathematics **119** (2002), 117–128.

[Pun01] A. P. Punnen, *The traveling salesman problem: new polynomial approximation algorithms and domination analysis*, Journal of Information and Optimization Sciences **22** (2001), 191–206.

[PvdV95] C.N. Potts and S.L. van de Velde, *Dynasearch-iterative local improvement by dynamic programming. part i. the traveling salesman problem*, Tech. report, University of Twente, The Netherlands, 1995.

[SD81] V.I. Sarvanov and N.N. Doroshko, *Approximate solution of the traveling salesman problem by a local algorithm with scanning neighborhoods of factorial cardinality in cubic time*, Software: Algorithms and Programs **31** (1981), 11–13.

[Sha98] Paul Shaw, *Using constraint programming and local search methods to solve vehicle routing problems*, Principles and Practice of Constraint Programming — CP98 (Michael Maher and Jean-Francois Puget, eds.), vol. 1520, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 417–431.

[Tai93] E. Taillard, *Parallel iterative search methods for vehicle routing problems*, Networks **23** (1993), 661–661.

[Tai03] Éric D. Taillard, *Heuristic methods for large centroid clustering problems*, Journal of Heuristics **9** (2003), 51–73.

[TBG+97] Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin, *A tabu search heuristic for the vehicle routing problem with soft time windows*, Transportation science **31** (1997), no. 2, 170–186.

[Tho88] P. M. Thompson, *Local search algorithms for vehicle routing and other combinatorial problems*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1988.

[TO89] Paul Michael Thompson and James B. Orlin, *The theory of cyclic transfers*, Tech. Report OR 200-89, Massachusetts Institute of Technology, Operations Research Center, 1989.

[TP93] Paul M. Thompson and Harilaos N. Psaraftis, *Cyclic transfer algorithms for multivehicle routing and scheduling problems*, Operations Research **41** (1993), no. 5, 935–946.

[Yeo97] Anders Yeo, *Large exponential neighbourhoods for the traveling salesman problem*, Tech. Report 47, Dept of Maths and CS, Odense University, Odense, Denmark, 1997.

[YIIG04] Mutsunori Yagiura, Shinji Iwasaki, Toshihide Ibaraki, and Fred Glover, *A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem*, Discrete Optimization **1** (2004), 87–98.