Noname manuscript No.
(will be inserted by the editor)

Constraint-Based Very Large-Scale Neighborhood Search

Sébastien Mouthuy, Pascal Van Hentenryck and Yves Deville

the date of receipt and acceptance should be inserted later

Abstract Very Large-Scale Neighborhood (VLSN) search is the idea of using neighborhoods of exponential size to find high-quality solutions to complex optimization problems efficiently. However, so far, VLSN algorithms are essentially described and implemented in terms of low-level implementation concepts, preventing code reuse and extensibility which are trademarks of constraint-programming systems. This paper aims at remedying this limitation and proposes a constraint-based VLSN (CBVLSN) framework to describe VLSNs declaratively and compositionally. Its main innovations are the concepts of cycle-consistent MoveGraphs and compositional moves which make it possible to specify an application in terms of constraints and objectives and to derive a dedicated VLSN algorithm automatically. The constraint-based VLSN framework has been prototyped in COMET and its efficiency is shown to be comparable to dedicated implementations.

1 Introduction

This paper is concerned with Very Large-Scale Neighborhood (VLSN) search, a class of local-search algorithms whose neighborhoods contain a large number of neighbors (usually exponential). By considering neighborhoods of exponential size, VLSN search often produces local optima of higher quality than polynomial-sized neighborhoods. These exponential neighborhoods are obtained by considering, as neighbors, configurations that can be reached by a set or a sequence of moves. The implementation of a VLSN algorithm must then handle two fundamental issues:

(1) How to search an exponential neighborhood in polynomial time and still obtain a high-quality neighbor?
(2) How to compute efficiently the impact of a set of moves without actually applying them?

Sébastien Mouthuy, Yves Deville
Université catholique de Louvain, Department of Computing Science and Engineering
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
E-mail: {Sebastien.Mouthuy,Yves.Deville}@uclouvain.be

Pascal Van Hentenryck
Brown University, Box 1910 Providence, RI 02912, USA
E-mail: pvh@cs.brown.edu

Traditionally, the first problem is solved by using dynamic programming or a modified shortest-path algorithm to explore the neighborhood. The second problem is addressed by designing dedicated algorithms that exploit the problem structure of specific applications.

VLSN algorithms have been successfully applied to a variety of NP-Hard problems such as the capacitated minimum spanning tree [1,2], exam timetabling [3,4,5], and block-to-train assignment [6]. See [7] for a survey of the main applications and VLSN approaches. Unfortunately, although they often share a common design, VLSN algorithms are almost always defined in terms of various data structures and specialized to the application at hand. This does not encourage reuse, extensibility, and experimentation.

This paper aims at remedying this difficulty and proposes a constraint-based VLSN (CBVLSN) framework that allows VLSN algorithms to be specified in terms of constraints and objectives, in the same way as local-search algorithms are expressed in constraint-based local search (CBLS) [8]. As a result, VLSN algorithms can now be specified by high-level models and support the traditional compositionality, reuse, and extensibility of constraint programming. The key technical contribution of the CBVLSN framework is *to demonstrate that VLSN algorithms can be synthesized automatically from constraints and objectives: What is needed is a natural extension of the CBLS interface for constraints and objectives to specify the input and output variables of every move.* Once input/output variables are specified, VLSN algorithms can be modeled and synthesized automatically by using the following four ideas:

(1) The model isolates a global constraint which captures the fundamental structure of the application and would be violated by moves individually.
(2) The synthesis produces a cycle-consistent MoveGraph, that is a graph whose cycles define the neighbors of the current solution. Each such neighbor preserves the satisfaction of the isolated global constraint.
(3) The synthesis only considers sequences of compositional moves, which are sequences such that it is possible to evaluate the effect of applying this sequence from the effects of its moves.
(4) The concept of input/output variables naturally enables the VLSN algorithm to automatically restrict attention to compositional moves.

The CBVLSN framework has been implemented and experimental results demonstrate the feasibility and benefits of the approach.

This paper is organized as follows. Section 2 presents some preliminary definitions and background. Section 3 presents the theoretical framework for constraint-based VLSNs. Section 4 introduces the concepts of input and output variables and Section 5 describes how the generic VLSN can be searched efficiently. Section 6 describes the architecture of the (Open Source) implementation of our framework in COMET. Section 7 illustrates how real-life problems can be solved by our constraint-based VLSN framework. Section 8 discusses the related works and Section 9 concludes the paper.

2 Preliminaries

This section defines some of the main concepts and notations used in the paper. It also introduces informally some of the concepts that are central to the VLSN framework specified in subsequent sections.

2.1 Combinatorial Optimization Problems

Let $\mathcal{X} = [X_1, X_2, \ldots, X_n]$ be a set of n variables taking their values in a domain D. We define an assignment as a function $\sigma : \mathcal{X} \rightarrow D$ that assigns a value to each variable. We denote the set of all possible assignments by Λ. A constraint is a function $\mathcal{C} : \Lambda \rightarrow \mathbb{N}$ giving the violation of a given assignment. A solution to a constraint $\mathcal{C}$ is an assignment σ satisfying $\mathcal{C}(\sigma) = 0$. An objective is a function $f : \Lambda \rightarrow \mathbb{Z}$ giving the evaluation of the quality of a given assignment. A Combinatorial Optimization Problem (COP) is a tuple $\mathcal{P} = \langle f, \mathcal{C}, \mathcal{X}, D \rangle$ and solving $\mathcal{P}$ requires finding a solution to $\mathcal{C}$ that minimizes f.

2.2 Constraint-Based Local Search

Constraint-based Local Search (CBLS) [8] is the idea of performing local search on high-level models. In CBLS, local-search algorithms are expressed in terms of two components: (1) a model which specifies the constraints and the objective and (2) a search algorithm which uses the constraints and objective to drive the search towards high-quality solutions. Hence, CBLS makes it possible to define local-search algorithms compositionally and to separate the model and the search components as in traditional constraint programming. *It is the goal of this paper to achieve a similar level of abstraction for VLSN algorithms.*

In CBLS, each constraint and objective is a differentiable invariant. Constraints incrementally maintain their violations and objectives their values after each iteration. In addition, each constraint and objective provides a differentiability Application Programming Interface (API) which evaluates the impact of local moves on these values. A move is a function $m : \Lambda \rightarrow \Lambda$. The set of all possible moves is problem-dependent and is denoted by $\mathcal{M}$. Given the current assignment σ, $\Delta_{\mathcal{C}}(m, \sigma)$ and $\Delta_f(m, \sigma)$ denote the changes induced by performing the move m on the violations of constraint $\mathcal{C}$ and on the value of objective f. These values are defined as follows:

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

$$\Delta_f(m, \sigma) = f(m(\sigma)) - f(\sigma).$$

2.3 Permutation Problems

As mentioned in the introduction, our VLSN framework relies on isolating a particular substructure of the application. Permutation constraints is one example of substructure arising in many applications (e.g., [9]). A permutation problem on the variables $\mathcal{X} = [X_1, \ldots, X_n]$ over the domain $D = \{1, \ldots, n\}$ seeks an assignment σ of the variables $\mathcal{X}$ such that the values of all the variables are distinct:

$$\sigma(X_i) \neq \sigma(X_j) \iff \forall i, j = 1, \ldots, n : i \neq j.$$

Definition 2.1 A **permutation constraint** on $\mathcal{X}$ is the function $\mathcal{C}_{perm} : \Lambda \rightarrow \mathbb{N}$ such that $\mathcal{C}_{perm}(\sigma) = 0$ if and only if σ assigns a permutation of D to the variables $\mathcal{X}$.

Example 2.1 The Traveling Salesman Problem (TSP) is perhaps the best known permutation problem. The TSP searches 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 $\langle f_{TSP}, \mathcal{C}_{perm}, X, D\rangle$. Many moves have been defined for this problem but we only consider assignment moves in this example. The move $assign(X_j, i)$ assigns the value i to variable X_j. More precisely, given an assignment σ, the move $assign(X_j, i)(\sigma)$ returns the assignment σ' with

$$\sigma'(X_k) = \begin{cases} i & \text{if } k = j \\ \sigma(X_k) & \text{otherwise} \end{cases}$$

The differentiation of such moves on the TSP is

$$\begin{aligned} \Delta_{f_{TSP}}(assign(X_j, i), \sigma) = &- c_{\sigma(X_{j-1}),\sigma(X_j)} - c_{\sigma(X_j),\sigma(X_{j+1})} \\ &+ c_{\sigma(X_{j-1}),\sigma(X_i))} + c_{\sigma(X_i),\sigma(X_{j+1})} \end{aligned} \tag{2.2}$$

where $X_0 = X_n$ and $X_{n+1} = X_1$. A single assignment move breaks the permutation structure of an assignment. For this reason, assignment moves are not used in standard local search approaches for solving the TSP, or permutation problems in general. Some VLSN algorithms for the TSP consider sequences of assignment moves that preserves the permutation structure. For instance, the sequence of moves

$$assign(X_1, \sigma(X_2)), assign(X_2, \sigma(X_3)), assign(X_3, \sigma(X_1))$$

preserves the permutation, although the individual moves break it. A key aspect of our VLSN framework is to ensure that only sequences of moves preserving the isolated substructure are considered as neighbors.

2.4 Partitioning Problems

A partition is another widely used substructure in VLSN algorithms. Partitioning problems are defined over set variables $\mathcal{X} = [S_1, \ldots, S_K]$ that represent subsets of $D = \{1, \ldots, n\}$.

Definition 2.2 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. $\forall i, j \in \{1, \ldots, K\} : \sigma(S_i) \cap \sigma(S_j) = \emptyset \iff i \neq j$
2. $\bigcup_{k=1}^{K} \sigma(S_k) = D.$

Example 2.2 The Generalized Assignment Problem (GAP) is a partitioning problem. Given a set of tasks $D = \{1, \ldots, n\}$ to be performed, the GAP seeks a partition of D into K machines. The variables $\mathcal{X} = [S_1, \ldots, S_K]$ represent the partition and S_k represents the set of tasks assigned to machine k. Each task i has a demand b_i and the machines have a capacity B. For each machine k, the sum of its demands cannot exceed its capacity, i.e.,

$$\sum_{i \in \sigma(S_k)} b_i \leq B \qquad \forall k = 1, \ldots, K$$

The violation of this capacity constraint is specified as follows:

$$\mathcal{C}_{GAP}(\sigma) = \sum_{k=1}^{K} \mathcal{C}_{CAPA}(k, \sigma) \tag{2.3}$$

where $\mathcal{C}_{CAPA}(k, \sigma) = \max\left(0, \sum_{i \in \sigma(S_k)} b_i - B\right)$. There is also a cost c_{ki} to assign task i to machine k and the objective to minimize is

$$f_{GAP}(\sigma) = \sum_{k=1}^{K} \left(\sum_{i \in \sigma(S_k)} c_{ki} \right) \tag{2.4}$$

The COP $\langle f_{GAP}, \mathcal{C}_{part} + \mathcal{C}_{GAP}, \mathcal{X}, 2^D \rangle$ specifies the Generalized Assignment Problem. We consider three types of moves for partitioning problems: $replace(S_k, i, j)$ 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. These moves are illustrated here below.

Example 2.3 Let the variable S and the assignment σ be such that $\sigma(S) = \{1, 2, 3, 4, 5\}$. The value assigned to variable S after having applied the move m on σ is denoted by $m(\sigma)(S)$. We have $insert(S, 6)(\sigma)(S) = \{1, 2, 3, 4, 5, 6\}$, $remove(S, 3)(\sigma)(S) = \{1, 2, 4, 5\}$ and $replace(S, 3, 6)(\sigma)(S) = \{1, 2, 4, 5, 6\}$.

The differentiation of these moves on the constraint and objective function is

m	$\Delta_{\mathcal{C}_{CAPA}}(m, \sigma)$	$\Delta_{f_{GAP}}(m, \sigma)$
$insert(S_k, i)$	$\max\left(0, \sum_{e \in \sigma(S_k)} b_e + b_i - B\right) - \mathcal{C}_{CAPA}(k, \sigma)$	$+c_{k,i}$
$remove(S_k, i)$	$\max\left(0, \sum_{e \in \sigma(S_k)} b_e - b_i - B\right) - \mathcal{C}_{CAPA}(k, \sigma)$	$-c_{k,i}$
$replace(S_k, i, j)$	$\max\left(0, \sum_{e \in \sigma(S_k)} b_e - b_j + b_i - B\right) - \mathcal{C}_{CAPA}(k, \sigma)$	$c_{k,i} - c_{k,j}$

(2.5)

Applying any of these moves breaks the partition structure of an assignment. A VLSN algorithm considers only sequences of such moves that preserve the partition structure. This is, for instance, the case of the sequence

$$remove(S_1, 2), replace(S_2, 2, 5), insert(S_3, 5).$$

Once again, a key contribution of our VLSN framework is to isolate such sequences automatically and efficiently. Moreover, we are interested in sequences that maintain the feasibility of $\mathcal{C}_{GAP}$ and improve the objective f_{GAP}.

2.5 Very Large-Scale Neighborhoods

The core idea of VLSN algorithms is to apply sequences of moves. If $\mathcal{M}$ denotes the set of moves, a VLSN algorithm selects, at each iteration, a sequence of moves $[m_1, \ldots, m_k]$ ($m_i \in \mathcal{M}$). In a first approximation, the neighborhood of an assignment σ for problem $\mathcal{P}$ is defined as

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

The main step of a VLSN algorithm is to find $\sigma' \in VLSN(\mathcal{P}, \sigma)$ such that $\mathcal{C}(\sigma') = 0$ and $f(\sigma')$ is minimal.

Since the size of the neighborhood is $2^{|\mathcal{M}|}$, a key aspect of VLSN algorithm is to find a way to explore that neighborhood efficiently. Note however that the richness of the neighborhood means that the local minima of a VLSN algorithm are often better than those of a pure local-search algorithm.

3 An Abstract Theory of Constraint-Based VLSN

In VLSN algorithms, neighbors of a solution are obtained by applying a sequence of moves and the size of the neighborhood is exponential. VLSN algorithms thus raise three fundamental issues:

(1) how to select several moves that are not interfering with each others?
(2) how to select a sequence of moves preserving the structural constraint?
(3) how to differentiate a sequence of moves effectively?

These issues are solved by imposing restrictions on the move sequences. In particular, a VLSN algorithm in our framework only selects moves that modify different variables (independence). It also selects only sequences of moves preserving the structural constraint, which is achieved through the concept of a *MoveGraph*. The third issue, efficient differentiation, is achieved by restricting attention to compositional moves. The rest of this section presents these concepts and progressively refines the definition of the VLSN neighborhood to address the above issues.

3.1 Sequential Composition of 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 modifies the current assignment. We thus define the sequential composition of several moves on an assignment. Intuitively, the sequential composition of moves m_1 and m_2 on assignation σ applies both moves on σ. If the resulting two assignations diverge on the value of a variable, the value resulting from move m_1 is chosen for this variable.

Definition 3.1 Given an assignment σ and two moves m_1 and m_2, the **sequential composition** $m_1|m_2$ 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 3.1 We consider assignments on variables X_1 and X_2 and 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 $m_2(\sigma)(X_2) = \sigma(X_1)$. Consider the assignment $\sigma = \{X_1 \to 2, X_2 \to 1\}$. We have $m_1(\sigma) = \{X_1 \to 3, X_2 \to 1\}$, $m_2(\sigma) = \{X_1 \to 2, X_2 \to 2\}$ and $m_1|m_2(\sigma) = \{X_1 \to 3, X_2 \to 2\}$. We also have $m_2|m_1(\sigma) = \{X_1 \to 3, X_2 \to 2\}$.

Proposition 3.1 *Sequential composition 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

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

On the other hand, we have

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

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

We can now define the sequential composition of a sequence of moves.

Definition 3.2 Given a sequence of moves $M = [m_1, \ldots, m_k]$ and an assignment σ, the sequential composition of M on σ is defined as

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

3.2 Move Independence

VLSN algorithms are often defined on independent moves.

Definition 3.3 A move m_1 is **independent** from a move m_2 wrt σ iff

$$\forall X_i \in \mathcal{X} : m_1(\sigma)(X_i) \neq X_i \Rightarrow m_2(\sigma)(X_i) = X_i.$$

A sequence of moves M is independent wrt σ if every move $m \in M$ is independent from every other move in M wrt σ.

Independent moves can be commuted, because the composition order has no effect on the resulting assignment.

Proposition 3.2 *The sequential composition of independent moves is commutative.*

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

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

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

Because by independence, we have

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

we obtain

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

By a similar reasoning, we obtain the same result for $m_2|m_1(\sigma)(X_i)$ and hence, for all variables X_i, we have $m_1|m_2(\sigma)(X_i) = m_2|m_1(\sigma)(X_i)$ for any assignment σ, i.e.,

$$\forall \sigma : m_1|m_2(\sigma) = m_2|m_1(\sigma).$$

This proves that, in 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. If M is a set of independent moves and σ is an assignment, we denote by $M(\sigma)$ the assignment obtained by the sequential composition of the moves in M on σ. As a result, we can refine our earlier definition of the VLSN neighborhood to become

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

Example 3.2 Consider the TSP. The move $assign(X_i, v)$ only modifies the value assigned to the variable X_i and is independent from the move $assign(X_k, w)$ if and only if $i \neq k$.

Example 3.3 Consider the GAP. The moves $replace(S_k, i, j)$ and $replace(S_m, j, i)$ are independent if and only if $k \neq m$. The moves $insert(S_1, 8)$ and $remove(S_1, 8)$ are independent if and only if $8 \notin S_1$.

3.3 Maintaining Structural Feasibility

We here address the second difficulty of selecting moves such that a structural constraint is not violated. We partition the constraints $\mathcal{C}$ of a 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 side-constraints. Typical examples of core constraints arising in VLSNs are permutation and partition constraints and moves are generally designed with these constraints in mind.

3.3.1 MoveGraphs

A VLSN algorithm deals with the feasibility of the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ differently. In particular, a VLSN considers atomic moves maintaining $\mathcal{C}_2$ but possibly violating $\mathcal{C}_1$, provided that the set of moves ensures that $\mathcal{C}_1$ is satisfied after application of the moves. *This is captured by the novel concept of a MoveGraph, which encapsulates the search of move sets that satisfy* $\mathcal{C}_1$. Informally speaking, edges in a MoveGraph represent moves and a cycle represents a set of moves maintaining the feasibility of $\mathcal{C}_1$ (even if some moves of such cycles may violate $\mathcal{C}_1$). The following definition of a MoveGraph will be instantiated next for various global constraints and various neighborhoods.

Definition 3.4 Given an assignment σ, a **MoveGraph** $MG(\sigma)$ is a labeled graph $\langle V, E, \eta \rangle$ where η is a function $E \rightarrow \mathcal{M}$. Given $E' \subseteq E$, we denote by $\eta(E') = \{\eta(i,j) | (i,j) \in E'\}$. A move $\eta(i,j)$ is also denoted by η_{ij}.

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

Definition 3.5 Given an assignment σ, a MoveGraph $MG(\sigma)$ is **cycle-consistent** wrt the constraint $\mathcal{C}_1$ if

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

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

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

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

Section 5.3 shows 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 reusable concept in a constraint-based framework. We now illustrate the concept of a MoveGraph on two important global constraints in VLSN research: permutation and partitioning constraints.

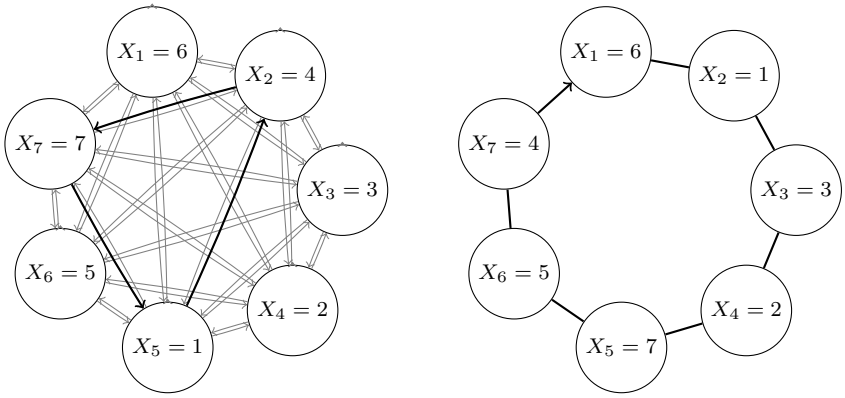
3.3.2 Permutation Problems

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

Definition 3.6 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 3.3 *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 3.4 Consider the TSP with $n = 7$ described in Example 2.1. Given the variables $\mathcal{X} = [X_1, \ldots, X_7]$ on domain $D = \{1, \ldots, 7\}$ and the current 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 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.

3.3.3 Partitioning Problems

We now present a MoveGraph for partitioning problems such as the Generalized Assignment Problem. The MoveGraph considers three types of moves: $replace(S_k, i, j)$ 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.

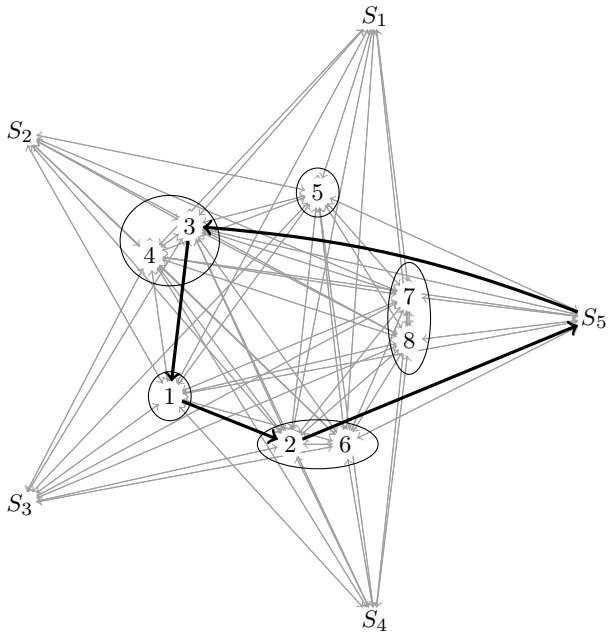
Definition 3.7 The MoveGraph $MG_{part}(\mathcal{X}, D, \sigma)$ for a partitioning problem and an assignment σ is the label graph $\langle V, E, \eta \rangle$ where

(1) $V = \mathcal{X} \cup D$,
(2) $E = \{(i, j) \in V \times V | i \in D \vee j \in D\}$,
(3) (a) For $i, j \in D, \eta(i, j) = replace(S_k, i, j)$ 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 $i \in \sigma(S_l)$.

Note that the semantics of the moves represented by an edge (S_k, i) does not depend on S_k. Variable S_k only appears to allow the moves to be performed in paths arriving at S_k. This MoveGraph is very similar to the graphs used in dedicated VLSN approaches for partitioning problems. The exact relation will be made clear in Section 3.4.2. This MoveGraph is cycle-consistent with respect to the partitioning constraint.

Proposition 3.4 *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 3.5 Consider the GAP introduced in Example 2.2 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: $replace(S_4,1,2)$, $insert(S_5,2)$, $remove(S_2,3)$ and $replace(S_3,3,1)$. 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 move violates it.

3.4 Ensuring Efficient Differentiation of a Set of Moves

Efficient differentiation of a set of moves is enabled through the concepts of compositional moves and improvement graph.

3.4.1 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 3.8 Given a 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

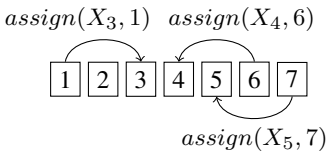
(1) $\Delta_{\mathcal{C}_2}(M,\sigma) = \sum_{m\in M} \Delta_{\mathcal{C}_2}(m,\sigma)$
(2) $\Delta_f\ (M,\sigma) = \sum_{m\in M} \Delta_f\ \ (m,\sigma)$

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,

$$VLSN_1(\mathcal{P},\sigma) = \{\eta(O)(\sigma) \mid O \text{ is a cycle in } MG(\sigma) \wedge \\ \eta(O) \text{ is independent and compositional wrt } \sigma\}.$$

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 3.6 Consider the TSP with $n = 6$ and illustrated below. Let the initial solution be $\sigma_0 = [1,2,3,4,5,6]$ with the cost $c_{12} + c_{23} + c_{34} + c_{45} + c_{56} + c_{61}$.



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

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

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

Example 3.7 Consider the GAP. For this problem, a set M of independent moves is necessarily compositional. Indeed we have

$$\Delta_{\mathcal{C}_{GAP}}(M,\sigma) = \sum_{k=1}^{K} \Delta_{\mathcal{C}_{CAPA}(S_k)}(M,\sigma) \tag{3.1}$$

and

$$\Delta_{\mathcal{C}_{CAPA(S_k)}}(M,\sigma) = \begin{cases} 0 & \text{if } \nexists m \in M \text{ modifying } S_k \\ \Delta_{\mathcal{C}_{CAPA(S_k)}}(m,\sigma) & \text{if there is a unique } m \in M \text{ modifying } S_k \end{cases} \tag{3.2}$$

Moreover, we have

$$\Delta_{\mathcal{C}_{CAPA(S_k)}}(m,\sigma) = \Delta_{\mathcal{C}_{GAP}}(m,\sigma) \tag{3.3}$$

with S_k being the variable modified by m. Thus

$$\Delta_{\mathcal{C}_{CAPA(S_k)}}(M,\sigma) = \sum_{m \in M} \Delta_{\mathcal{C}_{GAP}}(m,\sigma) \tag{3.4}$$

The same holds for f_{GAP} by decomposing it as a sum of k terms. We thus have that any set of independent moves is compositional for the GAP.

3.4.2 Improvement Graph

The neighborhood $VLSN_1(\mathcal{P},\sigma)$ can be searched efficiently through the concept of an improvement graph, which is built automatically from the MoveGraph. *Improvement graphs are the core of VLSN algorithms, and 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 3.9 Given a COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, a MoveGraph $G = \langle V, E, \eta\rangle$ and a solution 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 $VLSN_1(\mathcal{P},\sigma)$ can then be explored more efficiently by searching for cycles in the improvement graph, for the three following reasons:

1. Pruning the set of edges does not restrict the neighborhood. Indeed, the constraint $\mathcal{C}_2$ is satisfied by every solution σ. Thus $\Delta_{\mathcal{C}_2}(m,\sigma) \geq 0, \forall m \in \mathcal{M}$. So a set of compositional moves M satisfies $\mathcal{C}_2$ if and only if each single move $m \in M$ respects it ($\sum_{m\in M} \Delta_{\mathcal{C}_2}(m,\sigma) = 0 \iff \Delta_{\mathcal{C}_2}(m,\sigma) = 0, \forall m \in M$). Thus any compositional cycle O in the MoveGraph G such that $\Delta_{\mathcal{C}_2}(\eta(O),\sigma) = 0$ is also in the corresponding improvement graph.
2. The values $\Delta_f(m,\sigma)$ can be pre-computed, which means that the search for compositional cycles does not have to compute them repeatedly thanks to Definition 3.8.
3. The time-complexity of building the improvement graph is the same as scanning the full neighborhood in a standard local search. Indeed, 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. And this must also be done in standard local search algorithms.

3.4.3 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, but 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 3.5 *Given a 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 and η_{ij} be a move compositional with M. We have*

$$\Delta_f(\eta_{ij}, M(\sigma)) = \Delta_f(\eta_{ij}, \sigma) \tag{3.5}$$
$$\Delta_{\mathcal{C}_2}(\eta_{ij}, M(\sigma)) = \Delta_{\mathcal{C}_2}(\eta_{ij}, \sigma) \tag{3.6}$$

Proof

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

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

Proposition 3.5 shows that the presence and the cost of an edge (i, j) in the improvement graph are constant if for each move η_{ij} compositional with M. 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 η_{ij} represents another move (which could be possible as the function η depends on σ).

4 Automatic Derivation of Independent and Compositional Moves

Let us review what we have achieved in the previous section. 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 side-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 independent and compositional, can be computed very efficiently.

The only remaining issues are how to test compositionality and how to search the cyclic neighborhood. This section deals with the first issue. The second issue will be tackled in Section 5.

One possibility to test compositionality is to implement directly Definition 3.8. 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. *Input and output variables are two fundamental concepts used to derive a sufficient condition for a set of moves to be compositional and independent.*

First we define the output variables and how they can be used to compute independence in Section 4.1. We then introduce input variables in Section 4.2 and how they can be computed for a combination of differentiable invariants in Section 4.3. We describe how compositionality can be ensured based on the input variables in Section 4.4. We finally slightly restrict the cyclic VLSN to be search automatically and efficiently in Section 4.5.

4.1 Checking Independence Automatically

Independence can be checked by means of output variables.

Definition 4.1 Given a set of variables $\mathcal{X}$ and an assignment σ, the **output variables** of a move m, denoted by $\text{Var}^{\neq}(m, \sigma)$, is the set of the variables modified by applying the move m on σ: $\text{Var}^{\neq}(m, \sigma) = \{X_i \in \mathcal{X} : m(\sigma)(X_i) \neq \sigma(X_i)\}$.

We can then express independence in terms of output variables.

Definition 4.2 Given two moves m_1 and m_2 and an assignment σ, m_1 and m_2 are **variable-independent** wrt σ iff $\text{Var}^{\neq}(m_1, \sigma) \cap \text{Var}^{\neq}(m_2, \sigma) = \emptyset$.

Proposition 4.1 *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.*

4.2 Input Variables

Input variables are used to automatically ensure compositionality. We here define them after two small definitions. The first one captures when two assignments assign the same values to a subset of the variables.

Definition 4.3 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

$$\sigma_1(X_i) = \sigma_2(X_i) \qquad , \forall X_i \in X.$$

The next definition captures whether a set of variables allows to differentiate a function.

Definition 4.4 Given a function g, a move m and an assignment σ, a subset X of variables is **$\Delta_g(m, \sigma)$-complete** iff

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

for all assignments $\sigma' \in \Lambda$ that are X-equivalent wrt σ.

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 4.5 Given a set of variables $\mathcal{X}$, a move m, an assignment σ and a function $g : \Lambda \to \mathbb{Z}$, the **input variables** $Var^{<}(g, m, \sigma)$ is a smallest subset of $\mathcal{X}$ that is $\Delta_g(m, \sigma)$-complete.

Observe that a variable that is not in the expression of $\Delta_g(m, \sigma)$ cannot be an input variable. Thus we can state that the input variables $Var^{<}(g, m, \sigma)$ are at most the variables present in the expression of $\Delta_g(m, \sigma)$.

Example 4.1 Consider the TSP and the move $m = assign(X_j, i)$. The input variables $\text{Var}^{<}(f_{TSP}, m, \sigma) = \{X_{j-1}, X_j, X_{j+1}\}$. Indeed, these are the only three variables present in the expression of the variation of the move on the objective function (Equation (2.2)). It states that in order to compute the variation of the assign move $assign(X_j, i)$, we must know the value of the two adjacent cities of visit X_j.

Example 4.2 Consider the GAP and the moves $replace$, $insert$, or $remove$. From the array (2.5), there is only one input variable per move, namely the variable S_k modified by the move. Thus

$$\mathrm{Var}^{<}(f_{GAP}, m, \sigma) = \mathrm{Var}^{<}(\mathcal{C}_{GAP}, m, \sigma) = \mathrm{Var}^{\neq}(m, \sigma) \tag{4.1}$$

This means that only variable S_k must be considered to differentiate the constraint and the objective function wrt one move for this problem. Equation (4.1) states that

$$\mathrm{Var}^{<}(f_{GAP}, m, \sigma) \cup \mathrm{Var}^{<}(\mathcal{C}_{GAP}, m, \sigma) \subseteq \mathrm{Var}^{\neq}(m, \sigma)$$

and this explains why a set of independent moves is necessarily compositional for the GAP (Example 3.7).

Example 4.3 Consider the variables $\mathcal{X} = [X_1, X_2, X_3]$, the domain $D = \{0, 1\}$ and the function $f(\sigma) = \sigma(X_1) \cdot \sigma(X_2) \cdot \sigma(X_3)$. 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. It is possible to define a unique, but larger, set of input variables. For reasons that will become clear later, we are interested in having the smallest set of input variables possible.

4.3 Combinations of Differentiable Invariants

Once the input variables are defined for basic constraints and objectives, the input variables can also be synthesized for traditional logical and arithmetic operators.

Proposition 4.2 *The input variables can be determined for the sum, product or other operations $\diamond$ between multiple functions:*

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

Example 4.4 Consider the TSP. Let the function

$$element(X, Y, c) : \Lambda \to \mathbb{N} : element(X, Y, c)(\sigma) = c_{\sigma(X),\sigma(Y)}$$

where X, Y are two integer variables and c is a two-dimensional matrix. We define

$$f_{TSP'}(\sigma) = \sum_{i=1}^{n-1} element(X_i, X_{i+1}, c) + element(X_n, X_1, c) \tag{4.2}$$

Clearly the function $f_{TSP'}$ is equivalent to f_{TSP} although it is modeled using smaller functions. We have

$$\Delta_{element(X,Y,c)}(assign(X_j, i), \sigma) = \begin{cases} -c_{\sigma(X),\sigma(Y)} + c_{\sigma(X),i} & \text{if } X_j = Y \\ -c_{\sigma(X),\sigma(Y)} + c_{i,\sigma(Y)} & \text{if } X_j = X \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

Thus

$$\mathrm{Var}^{<}(element(X, Y, c), assign(X_j, i), \sigma) = \begin{cases} \{X, Y\} & \text{if } X_j = Y \\ \{X, Y\} & \text{if } X_j = X \\ \emptyset & \text{otherwise} \end{cases} \tag{4.4}$$

Let $m = assign(X_j, i)$,

$$\mathrm{Var}^{<}(f_{TSP'}, m, \sigma) = \bigcup_{i=1}^{n-1} \mathrm{Var}^{<}(element(X_i, X_{i+1}, c), m, \sigma) \cup \mathrm{Var}^{<}(element(X_n, X_1, c), m, \sigma) \tag{4.5}$$

$$= \{X_{j-1}, X_j\} \cup \{X_j, X_{j+1}\} \tag{4.6}$$

$$= \{X_{j-1}, X_j, X_{j+1}\} \tag{4.7}$$

$$= \mathrm{Var}^{<}(f_{TSP}, m, \sigma) \tag{4.8}$$

The decomposition of the objective function has the same input variables than the function considered globally in the case of the TSP.

This decomposability of functions is a critical benefit of our constraint-based approach. Indeed, when we model complex constraints and objective functions from a set of reusable basic constraints and objective functions, the CBVLSN search algorithm acts exactly as a dedicated implementation of the specific TSP objective. This indicates that complex VLSN algorithms can be built compositionally from primitive constraints and objectives.

4.4 Ensuring Compositionality Automatically

We now introduce a stricter notion of compositionality, based on the input variables.

Definition 4.6 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 4.3 *Given a COP $\langle f, \mathcal{C}_1 + \mathcal{C}_2, \mathcal{X}, D\rangle$, an assignment σ and a set of independent moves M, if 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 4.6 we have

$$\mathrm{Var}^{<}(f, m_j, \sigma) \cap \mathrm{Var}^{\neq}(m_i, \sigma) = \emptyset \qquad \forall i, j \in \{1, \ldots, k\} \text{ with } i \neq j \tag{4.9}$$

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 4.5 we obtain

$$\Delta_f(m_j, \sigma_{j-1}) = \Delta_f(m_j, \sigma) \tag{4.10}$$

Moreover we have

$$\begin{aligned}
\Delta_f(M_j, \sigma) &= f(M_j(\sigma)) - f(\sigma) && (4.11)\\
&= f(m_j(\sigma_{j-1})) - f(\sigma) && (4.12)\\
&= f(m_j(\sigma_{j-1})) - f(\sigma_{j-1}) + f(\sigma_{j-1}) - f(\sigma) && (4.13)\\
&= \Delta_f(m_j, \sigma_{j-1}) + f(M_{j-1}(\sigma)) - f(\sigma) && (4.14)\\
&= \Delta_f(m_j, \sigma_{j-1}) + \Delta_f(M_{j-1}, \sigma) && (4.15)
\end{aligned}$$

Thus, by (4.10), we obtain

$$\Delta_f(M_j, \sigma) = \Delta_f(M_{j-1}, \sigma) + \Delta_f(m_j, \sigma) \qquad \forall j = 2, \ldots, k \tag{4.16}$$

This recurrence formula leads to

$$\Delta_f(M_j, \sigma) = \sum_{i=1}^{j} \Delta_f(m_i, \sigma) \qquad \forall j = 1, \ldots, k \tag{4.17}$$

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

Example 4.5 Consider the GAP again. We illustrate variable-compositionality. Let the current solution σ be such that $\sum_{i \in e\sigma(S_k)} b_i = 10$ 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 individually and their output and input variables are both $\{S_k\}$. 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 individually. So our generic search algorithm knows these moves are not variable-compositional and will not select such moves together.

4.5 Searching VLSN Automatically

Input and output variables are two fundamental concepts: given a COP, they allow us to compute whether a set of moves is compositional and independent, without any additional knowledge from the user. This enables the implementation of arbitrarily complex VLSNs with side constraints 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,

$$\begin{aligned}
\mathrm{VLSN}_2(\mathcal{P}, \sigma) = \{\eta(O)(\sigma) \mid\ & O \text{ is a cycle in } MG(\sigma) \wedge \\
& \eta(O) \text{ is variable-independent} \\
& \text{and variables-compositional wrt } \sigma\}.
\end{aligned}$$

How to search $VLSN_2(\mathcal{P}, \sigma)$ is described hereafter.

5 Searching the cyclic VLSN

This sections presents how to search the neighborhood $VLSN_2(\mathcal{P}, \sigma)$ efficiently.

5.1 Global Search Algorithm

Algorithm 1 specifies the generic neighborhood exploration. It starts from a feasible solution and improves it by searching for a cycle in the improvement graph. This algorithm stops when no improving neighbors can be found.

Data: A solution σ_0, a MoveGraph $MG = \langle V, E, \eta \rangle$ and an objective function f.
Result: A solution σ respecting $\mathcal{C}_1$ and improving f wrt σ_0.

Let $\sigma = \sigma_0$;
Let $ig = IG(MG, \sigma)$;
while *true* do
    Search for a cycle $O \in ig$ such that $\eta(O)$ is a set of variable-independent and variable-compositional moves;
    if *no such cycle found or* $\sum_{(i,j)\in C} \Delta_f(\eta_{ij}, \sigma) \geq 0$ then break;
    Apply the set of moves: $\sigma = \eta(O)(\sigma)$;
    update $ig = IG(MG, \sigma)$;

Algorithm 1: Generic CBVLSN search algorithm. Given the current assignment and a MoveGraph MG, this algorithm explores the VLSN described by MG and returns a solution improving the objective function.

The remainder of this section describes the different steps of this algorithm. Namely (1) how we compute the independence and compositionality of a set of moves (2) how we search for a cycle in the improvement graph, and (3) how the improvement graph can be incrementally updated.

5.2 Checking Variable-Compositionality and Variable-Independence:

Checking variable-compositionality and variable-independence is a crucial step in the algorithm. We here describe how this check can be efficiently computed. Let M be a set of moves and σ be an assignment, checking whether a move m is variable-compositional and variable-independent wrt M can be done in $\mathcal{O}(|M| \cdot o_V + o_V)$, where o_V is an upper bound on 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. Let $inputMarked$ and $outputMarked$ be Boolean arrays both indexed by the variables in $\mathcal{X}$. The cells of these arrays corresponding to all variables in $\mathrm{Var}^{<}(m, \sigma)$ and $\mathrm{Var}^{\neq}(m, \sigma)$ are set to true. This can be done in $\mathcal{O}(o_V)$ where o_V is an upper bound on the number of input and output variables per move. Marking all the nodes in M can thus be done in $\mathcal{O}(|M| \cdot o_V)$.

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 $\mathrm{Var}^{\neq}(m, \sigma)$ is marked in both arrays, and if no input variables is marked in $outputMarked$. This check can be done in $\mathcal{O}(o_V)$.

5.3 Searching for cycles

This section describes how to search the cyclic neighborhood $VLSN_2(\mathcal{P}, \sigma)$. We start with a negative result.

Proposition 5.1 *The problem of finding the best candidate in $VLSN_2(\mathcal{P}, \sigma)$ is NP-Hard.*

Proof In order to prove that the general problem of searching $VLSN_2(\mathcal{P}, \sigma)$ is NP-Hard, we prove that in the particular case of the Generalized Assignment Problem, finding the best candidate in $VLSN_2(\mathcal{P}, \sigma)$ is NP-Hard. This can be proven by showing the equivalence between this last problem and the Cycle Through Distinct Subpartition Problem (CTDSP), that has been proven to be NP-Hard [10]. The CTDSP is the usual subproblem to be solved in cyclic VLSN dedicated to partitioning problems.

First, consider the MoveGraph for partitioning problems described in Section 3.3.3. Note that any $insert$, $remove$ or $replace$ move only modifies one variable. A move modifying the variable S_k only affects the k^{th} term of the constraint and objective functions of the GAP specified in Equations (2.3) and (2.4). Thus two moves modifying two different variables, will affect two different terms in (2.3) and (2.4), and will be compositional. Thus any set of independent moves is compositional for this particular problem.

Now, given a solution σ of the GAP, we define the collection of sets $[V_0, \ldots, V_{K-1}]$ where $V_k = \sigma(S_k) \cup \{S_k\}$ for all $k = 0, \ldots, K-1$. Because $[\sigma(S_0), \ldots, \sigma(S_{K-1})]$ represents a partition of D, the collection $[V_0, \ldots, V_{K-1}]$ is a partition of V.

Consider the MoveGraph $MG_{part}(\sigma) = (V = \mathcal{X} \cup D, E, \eta)$ defined in Section 3.3.3. Because of the definition of η, the variable modified by the move η_{ij} is only function of j: if $j \in \mathcal{X}$, then applying η_{ij} will modify the variable j, and if $j \in D$, then the variable being modified is the variable S_k containing the element j. Thus finding a cycle in $MG_{part}(\sigma)$ is equivalent to finding a cycle in this same graph that has at most one element of each of the subpartitions $V_0, \ldots, V_{K-1}$. This problem is called the Cycle Through Distinct Subpartition problem and has been proven to be NP-Hard in [10].

The general problem of searching the best candidate in $VLSN_2(\mathcal{P}, \sigma)$ is thus NP-Hard. □

The complexity of searching the best candidate in $VLSN_2(\mathcal{P}, \sigma)$ contrasts with the polynomial-time complexity of searching the minimum-cost cycle in a graph. As a result, traditional VLSN algorithms abandon completeness for polynomial-time behavior: They typically find a high-quality neighbor by using a heuristic to the Cycle Through Distinct Subpartition problem [1].

We follow a similar approach and our constraint-based VLSN framework uses a variation of that algorithm to search $VLSN_2(\mathcal{P}, \sigma)$. The algorithm, based on the label-correcting algorithm [11], is depicted in Algorithm 2 and handles edges with negative weight. The label-correcting algorithm builds a shortest path tree rooted at a start node s. This tree contains the shortest paths from s to any other node of the graph, except if no such path has been found. We use $P[i]$ to denote the shortest path from s to i in the shortest path tree (encoded by the array $pred$). Algorithm 2 maintains a list containing all nodes whose outgoing edges may improve this shortest path tree. The algorithm pops the nodes in this list, examines their outgoing edges and updates the shortest path tree accordingly. We restrict this algorithm to extend a path (in the shortest path tree) with an edge only if the corresponding moves are independent and compositional. It is similar to the algorithm presented in [1] except that here we also check for compositionality.

The complexity of Algorithm 2 depends on the implementation of `LIST`. If we use a queue (FIFO), then Algorithm 2 achieves a polynomial-time complexity.

Proposition 5.2 *Given a 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*

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

Algorithm 2: The Heuristic Shortest-Path Algorithm to Search for Independent and Compositional Cycles in an Improvement Graph. The label-correcting algorithm for searching for paths [11] is represented in non-bold font. The added lines are in bold. These lines allow to return a cycle only if it represents a set of independent and compositional moves.

path tree. The time-complexity of Algorithm 2 is $\mathcal{O}(nU^2o_V + mU(U + o_V))$ if using a FIFO implementation for `LIST`.

Proof First note that when popping a couple (k, i) from `LIST`, only couples of the form $(k + 1, j)$ are 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 $k' > k$ are 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 on the value of k. The while loop 3-12 is thus executed at most nU times.

Inside the while loop, marking the nodes (line 5) and edges in $P[i]$ can be done in $\mathcal{O}(U \cdot o_V)$ (Section 4 describes how). Checking independence and compositionality of the corresponding moves is done during the marking. So the complexity of lines 5-6 is $\mathcal{O}(nU^2o_V)$. 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 8 takes $\mathcal{O}(U)$, checking whether η_{ij} can be added to $P[i]$ in $\mathcal{O}(o_V)$ and all operations in lines 10-12 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 $\mathcal{O}(nU^2o_V + mU(U + o_V))$. □

This complexity is considerably lower in practice. 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 2 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 complexity as the algorithm presented in [1]. Indeed, for this problem, our algorithm performs exactly the same operations as in [1].

Figure 5.1 illustrates why the algorithm checks for independence and compositionality twice and why the algorithm is incomplete. Consider Algorithm 2 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

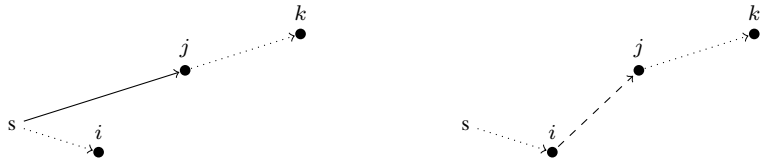


Fig. 5.1 Illustrating the Behavior of the Algorithm.

tree because the path $[(s,i),(i,j)]$ is independent and compositional (line 9). This implicitly 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 2 is incomplete, since it has lost track of the path from s to k going through j.

5.4 Update of the Improvement Graph

Once a set M of moves is selected and applied, the improvement graph must be recomputed. Section 3.4.3 showed that only moves non-compositional with M have to be reconsidered during this update. As Proposition 4.3 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 $conflict(M,\sigma) = \{(i,j) \in E : \mathrm{Var}^{<}(\eta_{ij},\sigma) \cap \mathrm{Var}^{\neq}(M,\sigma) \neq \emptyset\}$ have to be reconsidered.

6 Implementation

Our CBVLSN framework has been prototyped in COMET. We here present the general architecture and the main interfaces. The complete prototype is Open Source and is available at becool.info.ucl.ac.be.

The CBLS Interface The CBLS interface is extended to include input and output variables. Listing 1 depicts the CBLS interface for differential invariants in partitioning problems. In addition to the traditional CBLS interface, which supports the differentiation of replace, insert, and remove moves, the interface also contains methods to collect output variables (line 4) and input variables (lines 6, 8, and 10). The methods receive an object `VarCollector` which contains two sets for accumulating input and output variables of a set of moves. This interface makes VLSN algorithms generic: they do not rely on the semantics of the constraints, but only rely on their interface. This interface can then be used generically to compute independence and compositionality.

```
1  interface PartitionDifferentialInvariant<VLSN> {
2    Solver<VLSN> getLocalSolver();
3    var{int} value();
4    int getReplaceDelta(var{set{int}} S,int i,int j);
5    void getReplaceInputVariables(var{set{int}} S,int i,int j,
6                                  VarCollector t);
7    void getReplaceOutputVariables(var{set{int}} S,int i,int j,
```

```
8                                   VarCollector t);
9     int getInsertDelta(var{set{int}} S,int i);
10    void getInsertInputVariables(var{set{int}} S,int i,
11                                 VarCollector t);
12    void getInsertOutputVariables(var{set{int}} S,int i,
13                                 VarCollector t);
14    int getRemoveDelta(var{set{int}} S,int j);
15    void getRemoveInputVariables(var{set{int}} S,int j,
16                                 VarCollector t);
17    void getRemoveOutputVariables(var{set{int}} S,int j,
18                                 VarCollector t);
19  }
```

Listing 1 Interface for Differential Invariants.

Searching for Compositional Cycles To search for compositional cycles, our implementation uses the enhanced version of the label-correcting algorithm depicted in Algorithm 2. Note that the algorithm operates on the improvement graph, which is systematically derived from the MoveGraph. Several MoveGraphs are predefined in our current implementation, but all MoveGraphs implement the same interface and can be defined by users as we now discuss.

Defining new MoveGraphs In constraint-based VLSN, a cyclic VLSN is entirely defined by a MoveGraph. All MoveGraphs implement the interface depicted in Listing 2 which encodes nodes as integers for simplicity. The interface provides several important methods. Method `isMove(i,j)` determines if edge (i, j) corresponds to a move. If m is the move associated with edge (i, j), method `applyMove(i,j)` applies m on the current solution; method `isFeasibleMove(i,j)` determines if m is feasible and is used for constructing and updating the improvement graph; method `getDeltaMove(i,j)` differentiates m; method `getInputVariables(i,j,t)` collects the input variables of m and `getOutputVariables(i,j,t)` collects the output variables of m.

```
1  interface MoveGraph<VLSN> {
2    range getNodes();
3    bool isMove(int i,int j);
4    void applyMove(int i,int j);
5    bool isFeasibleMove(int i,int j);
6    int getMoveDelta(int i,int j);
7    void getInputVariables(int i,int j,VarCollector t);
8    void getOutputVariables(int i,int j,VarCollector t);
9  }
```

Listing 2 Interface for defining a MoveGraph

7 Models and Experimental Results

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 improve them. The experimental results presented here are a proof of concept of our theoretical framework. The first application is particularly interesting: It illustrates why compositionality, which is obtained automatically in constraint-based VLSN, is beneficial in practice.

The second application was considered because it is the only problem for which detailed experimental results of a dedicated VLSN implementation are available.

7.1 The Capacitated Examination Timetabling Problem

The Capacitated Examination Timetabling Problem (CETP) is a real-life problem encountered in universities. The goal of the CEPT is to partition n exams into K consecutive time slots $S = [S_1, \ldots, S_K]$, each slot taking place on a specific day, subject to the following hard constraints: (1) There are no students taking two exams scheduled in the same time slot (exclusion constraint), (2) 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 number of students taking exams i and j are given by the matrix $(c_{ij}) \in \mathbb{N}^{n \times n}$. The objective is to minimize the number of students having two exams the same day in two consecutive time slots. If we denote the day of time slot k by day_k, the objective function can be formulated as follows:

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

Thus the CEPT can be defined as the COP $\langle f, \mathcal{C}_{part} + \mathcal{C}_2, S, 2^N \rangle$, where $\mathcal{C}_{part}$ is the global partition constraint, $\mathcal{C}_2$ is the capacity plus the exclusion constraints, and $N = \{1, \ldots, n\}$. This problem can be implemented using our framework as follows.

The model is illustrated in Listing 3. The variable `p` represents a partition of the elements $\{1, \ldots, n\}$ into K sets. The differentiable objective `SubsetSum(X,Y,c)` represents the function $\sum_{i \in X, j \in Y} c_{ij}$. The input variables are empty for a move not modifying `X` nor `Y`, as for such moves the differentiation is zero. For a move modifying `X` or `Y`, the input variables are at most $\{X, Y\}$. The input variables of a move modifying a variable S_k wrt the aggregate sum `f` are thus

$$\mathrm{Var}^{<}(f, m, \sigma) = \left\{ S_i \in S \middle| |i - k| \leq 1 \text{ and } day_k = day_i \right\}$$

which would be exactly the input variables if f would have been modeled as a global function. The constraint system `Cs` contains all the constraints $\mathcal{C}_2$ of the problem: capacity and exclusion constraints. Posting these constraints for each set variables separately also leads to the same definition of input variables as using global constraints on the entire partition.

```
1  Solver<VLSN> m();
2  Partition<VLSN> p(m,1..n,1..K);
3  var{set{int}}[] S = p.getVariables();
4  CETPFunction f =
5     sum(k in 1..K-1 : day[k]==day[k+1]) SubsetSum(S[k],S[k+1],c);
6  PartitionConstraintSystem<VLSN> Cs(m);
7  forall(k in 1..K){
8    Cs.post(CapacityConstraint (S[k],D));
9    Cs.post(ExclusiveConstraint(S[k],c));
10 }
11 vs.close();
```

Listing 3 Model for the CETP.

The Search We used a GRASP (Greedy Randomized Adaptive Search Procedure) procedure such as described in [1]. First an initial solution is computed. Then the search iteratively 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.

The initial solution is computed as in [5], based on the saturation heuristic first described in [12]. This allows us to compare our approach with [5]. We here describe this heuristic in terms of the CEPT. The saturation degree heuristic iteratively selects exams and assigns them to periods. Given an exam i, let CE_i be the set of exams conflicting with i: $CE_i = \{j \in \{1,\ldots,n\} | c_{ij} > 0\}$ and let p_i be the timeslot assigned to an assigned exam i. At each iteration, the heuristic selects an unassigned exam with the greatest number of different timeslots assigned to exams in CE_i. The ties are broken by selecting the unassigned exam with the greatest set of exam in CE_i that have already been assigned. More formally, the unassigned exams i are iteratevely selected by decreasing lexicographical value of

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

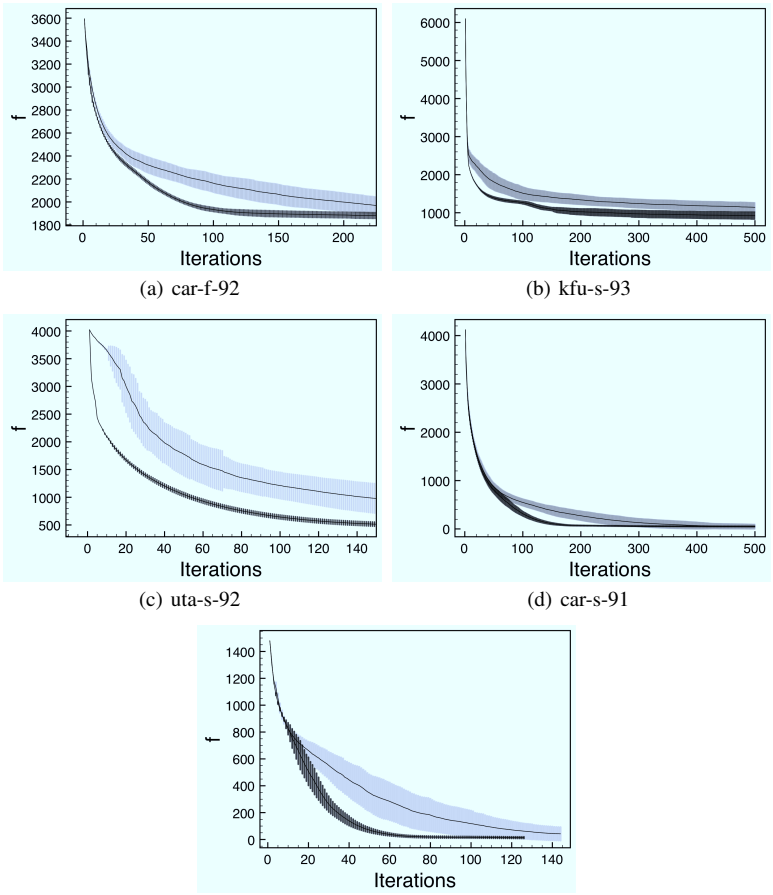
In the code, line 1 computes an initial solution. Line 2 constructs the MoveGraph and line 3 derives its associated improvement graph. In line 5, the best move found by the cyclic search algorithm is obtained: It encapsulates the actual move and the improvement graph (and thus the MoveGraph) to update the improvements when applied. It also can be differentiated as shown in line 6. Observe how the search is completely separated from the model and could thus be used for any other partitioning problems without any modification.

```
1    p.initialize();
2    PartitionExchangeMoveGraph<VLSN> mg(vs,p,Obj,Cs);
3    ImprovementGraph<VLSN> ig(mg);
4    while (System.getCPUTime() < timeLimit ){
5      Move<VLSN> M = ig.getBestCycle();
6      if (M == null || M.getDelta() > 0) p.initialize();
7      else M.apply();
8    }
```

Listing 4 Search for the CMST

Computational Comparison Hard instances for this problem are available [13]. Several works solved this problem by designing a dedicated VLSN approach and obtained the best known solutions to some of these instances [3,4,5]. They all search for an independent cycle (i.e., a cycle with no pair of moves modifying the same set variable). Such a neighborhood does not consider compositionality, leading thus to the possible selection of a negative cycle in the improvement graph that represents a set of moves degrading the objective function that must then be rejected. In order to compare both neighborhoods, 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



(a) car-f-92 (b) kfu-s-93

(c) uta-s-92 (d) car-s-91

(e) tre-s-92

Fig. 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) 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 largest error area around the bright curve).

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 derived compositionally from primitive constraints and objectives.

New best solutions Our constraint-based VLSN and the GRASP procedure found new best solutions to the CETP by only changing how the initial solutions were built. We here describe this change. In order to assign the exams to the timeslots, we select an exam not already assigned having the larger 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 con-

flicting exams. We thus select the exam i with the greater lexicographical value

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

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 repetitions of the construction process, a good initial solution is typically found.

Instance		Merlot et al.[14]	AAB[5]	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
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 Experimental results for the CEPT, using a new procedure to generate initial solutions. Our CBVLSN algorithm improves two instances (CAR-S-91, UTA-S-92), and find the optimal solution for a third one (TRE-S-92: a solution with a zero cost cannot be improved). 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 improve the generated initial solutions. Reported times have not been converted to account for different computing resources.

Our concise CBVLSN model and search for the CETP improves the best solution found for two of the data sets considered in [5], and obtain the optimal solution for the data set TRE-S-92 (Table 7.1). We compared our algorithm to [5] and [14] that are the two algorithms computing the best known solutions on these data sets. Unfortunately our new procedure computing the initial solutions didn't find a feasible initial partition for the fifth instance CAR-F-92. This instance has the greatest proportion of conflicting exams, that may explain why our procedure cannot find initial solution to the exclusion constraint.

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) .

7.2 The Capacitated Minimum Spanning Tree

The Capacitated Minimum Spanning Tree (CMST) is a communication problem. We are given a matrix $d \in \mathbb{R}^{n+1\times n+1}$ specifying the distance between all pairs of terminals and between all the 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 $S = [S_1, \ldots, S_K]$ of the n terminals respecting the capacity constraint $\sum_{i\in S_k} b_i \leq D, \forall k = 1, 2, \ldots, K$ and minimizing the following objective function

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

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 $\langle f, \mathcal{C}_{part} + \mathcal{C}_2, S, 2^N\rangle$, where $\mathcal{C}_{part}$ is the global partition constraint, $\mathcal{C}_2$ is the capacity constraint, and $N = \{1, \ldots, n\}$. We refer the reader to [1,2] for extensive references about the CMST.

A VLSN for the CMST was proposed in [1,2] 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 it is hardcoded in their search algorithm. It is thus easy to implement their algorithm in our constraint-based framework as follows.

The Model is illustrated in Listing 5 and it is very similar to the model for the CEPT. The differentiable objective `mst` computes the cost of the minimum spanning trees for a set of nodes, and `Obj` represents the cost of a partition for this problem. The constraint system `Cs` contains the capacity constraint of the problem.

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

Listing 5 Model for the CMST

The search The search component uses the same GRASP approach as the CEPT. The code in Listing 4 is thus reused for this new problem. We compute the initial solution as in [1] by using a randomized version of the greedy algorithm proposed in [15]. 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 [1].

New Neighborhood The VLSN algorithm presented above only exchanges single values between set variables. In [2], a more complex neighborhood is presented, in which subtrees of the current solution and/or single values can be exchanged among different subsets as

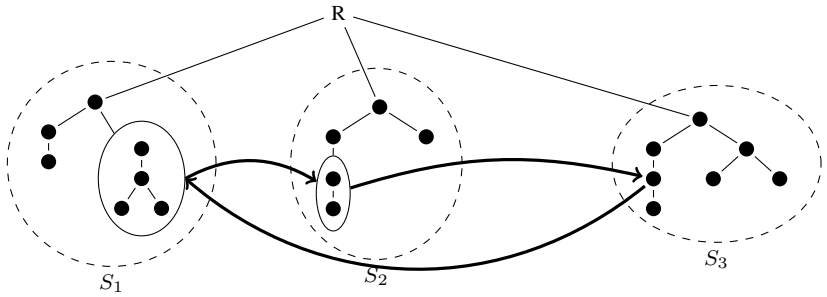


Fig. 7.2 Composite Cyclic Exchange for the CMST.

illustrated in Figure 7.2. Our abstractions allow to implement this new neighborhood by simply implementing a new MoveGraph. There is no need to modify the model or the search algorithm provided that the CBLS API supports the moves.

Computational Comparison The extensive computational study in [1,2] allows us to illustrate the advantages and drawbacks in terms of modularity and efficiency of implementing VLSN search algorithms using our CBVLSN framework. We implemented the same algorithm presented in [1] using our abstractions. We run our algorithm on the same instances and the initial solutions are computed exactly as in [1]. The GRASP is tuned with the same parameters (Time limit of 200 seconds, application of the first improving cycle found). The experimental analysis presented in [1] (denoted Du in Table 7.2) was made on Pentium 1,4 GHz with 512MB of memory. Our experiments (denoted MDVH) were performed 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 7.2 assesses that we reproduced the same VLSN algorithm as in [1]: both implementations behave similarly. They compute solutions of almost equivalent quality and perform roughly the same number of iterations per run. The slight differences in solutions and number of iterations are certainly due to some minor implementation details that were not described in [1]. The last column shows our implementation is about 4 times slower than the dedicated implementation of [1] (the difference of computers used is already taken into account in Table 7.2), which is not surprising since our abstractions are built on top of COMET which is an interpreted programming language. Any simple algorithm generally runs four time slower on COMET than if it is implemented in C++. Supporting the abstractions directly in the core of COMET will remove this gap.

8 Related Work

Many fast dedicated VLSN algorithms exist for solving different problems such as Capacitated Minimum Spanning Tree [1,2], exam timetabling [3,4,5], and block-to-train assignment [6]. All these approaches focus on a given problem to define and specify a specific VLSN. There are two major differences between the algorithm they use to search the cyclic neighborhood and ours (Algorithm 2): they do not check for compositionality, and the check for independence is not generic wrt the constraints and objectives of the problem. This makes their algorithms difficult to extend when considering either the addition of a constraint to the model or the definition of a new VLSN. Cyclic neighborhoods were first studied in [10] and

Instance	Q	Avg value		Nb of iter		Time per iter		Time
		MDVH	Du	MDVH	Du	MDVH	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 7.2 Experimental comparison of our implementation with [1]. 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.

applied to the vehicle routing problems in [16, 17]. The idea of defining VLSN as the composition of several independent moves emerged in [18]. They solved a vehicle routing problem and defined the condition two moves have to respect for their variation to be additive for this specific problem. Then they designed search algorithms to compute a set of moves satisfying this property, generally based on problem-specific improvement graphs. Ejection chains [19] can also be considered as compounding moves.

Some papers also design a generic methodology using dynamic programming or grammars to define new VLSN [20, 21, 22]. This allows to quickly define a VLSN for permutation problems and use a generic algorithm to search it. However, this approach operates at a much lower 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 to ensure independence and compositionality.

Recent work also relies on the variables to select moves that may be applied together [23]. 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. There are two main differences between their approach and ours. 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 do not satisfy these severe restrictions.

Large Neighborhood Search (LNS) can also be considered as VLSN. In LNS, at each iteration, a subset of the variables are unassigned and the neighborhoods are defined as the set of solution that can be obtained by reassigning these *free* variables. The neighbors are searched by using a problem-generic solver such as a Constraint Programming or a Mixed Integer Programming solver. The set of solutions that can be reached may be very big, and LNS can thus be categorized as VLSN.

In this work, we focused on VLSN that are defined as solutions that can be reached by performing a set of simple moves. The differences between the VLSN studied in this paper and LNS are twofold. First, in this paper, we apply a set of independent and compositional

moves. These moves modify variables that are not related. This is the key to be able to differentiate a set of moves. In LNS, generally, the stategy is the opposite: we select variables that are in some way *linked*. This leads to better results [24].

Second, the VLSN studied in this paper have more structure than the neighborhoods defined in LNS. This enables to modify polynomial-time algorithms to search VLSN very efficiently. In LNS, the neighborhoods are searched by problem-generic solvers. The algorithm selecting the neighbors may thus be less time-efficient. However they can handle any type of constraints or objective functions, which can lead to a generic LNS implementation [25].

9 Conclusion

This paper showed that the key idea behind CBLS can be naturally extended to VLSN search, a class of local search algorithms using neighborhoods of exponential size but whose good neighbors can be computed in polynomial time. By enhancing the CBLS interface with the simple concepts of input and output variables, we showed that a VLSN search can be systematically derived from a MoveGraph defining the moves to consider. The edges of the MoveGraph represent moves and its cycles capture a set of moves that maintain the feasibility of a global constraint representing the core structure of the application (e.g., a permutation or a partitioning). Moreover, by restricting attention to compositional moves, good neighbors in the VLSN can be found in polynomial time and the concepts of input and output variables provide the tool to detect if a set of moves is compositional. Note also that a variety of MoveGraphs can be provided to modelers, while implementers can implement their own through a well-defined interface.

This theoretical framework for constraint-based VLSNs transfers the benefits of CBLS, e.g., high-level modeling, separation of model and search, extensibility, and reuse, to VLSNs. A constraint-based VLSN framework will then provide a library of MoveGraphs, differential invariants that can be extended by users if necessary and generic VLSN search algorithms. These contributions make it possible to define VLSN models at a high level of abstraction and to include idiosyncratic constraints naturally.

The paper described the architecture of the implementation of the framework in COMET and presented experimental results demonstrating the feasibility of the approach on two significant problems. The complete prototype is Open Source and is available at becool.info.ucl.ac.be.

Our current work is focused on complex vehicle routing applications, where VLSNs bring significant benefits, and the integration of this framework into the core of COMET. We also investigate VLSN search algorithms that differentiate the model on-the-fly instead of using improvement graphs.

Acknowledgements

The first author is supported by the belgian FNRS ('Aspirant'). This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research).

References

1. R. K. Ahuja, J. B. Orlin, and D. Sharma, "Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem," *Mathematical Programming*, vol. 91, pp. 71–97, Oct. 2001.
2. R. K. Ahuja, J. B. Orlin, and D. Sharma, "A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem," *Operations Research Letters*, vol. 31, pp. 185–194, May 2003.
3. S. Abdullah, S. Ahmadi, E. Burke, M. Dror, and B. McCollum, "A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem," *Journal of the Operational Research Society*, vol. 58, pp. 1494–1502, Nov. 2007.
4. S. Abdullah, S. Ahmadi, E. Burke, and M. Dror, "Applying ahuja-orlin's large neighborhood for constructing examination timetabling solution," in *Proceedings of the Fifth International Conference on the Practice and Theory of Automated Timetabling*, no. 3616 in Lecture Notes in Computer Science, pp. 413–420, Springer, 2004.
5. S. Abdullah, S. Ahmadi, E. Burke, and M. Dror, "Investigating ahuja-orlin's large neighbourhood search approach for examination timetabling," *OR Spectrum*, vol. 29, pp. 351–372, Apr. 2007.
6. K. C. Jha, R. K. Ahuja, and G. Şahin, "New approaches for solving the block-to-train assignment problem," *Networks*, vol. 51, no. 1, pp. 48–62, 2008.
7. R. K. Ahuja, Özlem Ergun, J. B. Orlin, and A. P. Punnen, "A survey of very large-scale neighborhood search techniques," *Discrete Appl. Math.*, vol. 123, no. 1-3, pp. 75–102, 2002.
8. P. V. Hentenryck and L. Michel, *Constraint-Based Local Search*. MIT Press, Oct. 2005.
9. C. Solnon, "Solving permutation constraint satisfaction problems with artificial ants," in *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, (Berlin), pp. 118–122, August 2000.
10. P. M. Thompson and J. B. Orlin, "The theory of cyclic transfers," Tech. Rep. OR 200-89, Massachusetts Institute of Technology, Operations Research Center, 1989.
11. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, united states ed ed., Feb. 1993.
12. D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, pp. 251–256, Apr. 1979. ACM ID: 359101.
13. M. Carter and G. Laporte, "Recent developments in practical examination timetabling," *Practice and Theory of Automated Timetabling*, pp. 1–21, 1996.
14. L. Merlot, N. Boland, B. Hughes, and P. Stuckey, "A hybrid algorithm for the examination timetabling problem," *Practice and Theory of Automated Timetabling IV*, pp. 207–231, 2003.
15. L. R. Esau and K. C. Williams, "On teleprocessing system design: part II a method for approximating the optimal network," *IBM Systems Journal*, vol. 5, no. 3, pp. 142–147, 1966.
16. P. M. Thompson, *Local search algorithms for vehicle routing and other combinatorial problems*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1988.
17. P. M. Thompson and H. N. Psaraftis, "Cyclic transfer algorithms for multivehicle routing and scheduling problems," *Operations Research*, vol. 41, pp. 935–946, sep 1993.
18. O. Ergun, J. B. Orlin, and A. Steele-Feldman, "Creating very large scale neighborhoods out of smaller ones by compounding moves: A study on the vehicle routing problem," Tech. Rep. 4393-02, MIT Sloan School of Management, oct 2002.
19. F. Glover, "Ejection chains with combinatorial leverage for the tsp," *Discrete Applied Mathematics*, vol. 65, pp. 223–253, 1996.
20. O. Ergun and J. B. Orlin, "A dynamic programming methodology in very large scale neighborhood search applied to the traveling salesman problem," *Discrete Optimization*, vol. 3, pp. 78–85, Mar. 2006.
21. 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*, vol. 3509/2005, pp. 437–451, 2005.
22. R. E. Burkard, V. G. Deineko, and G. J. Woeginger, "The travelling salesman and the pq-tree," *Mathematics of Operations Research*, vol. 23, pp. 613–623, 1998.
23. T. Benoist, "Characterization and automation of matching-based neigborhood," in *CPAIOR'10*, 2010.
24. L. Perron, P. Shaw, and V. Furnon, "Propagation guided large neighborhood search," in *Principles and Practice of Constraint Programming – CP 2004* (M. Wallace, ed.), vol. 3258, pp. 468–481, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.
25. R. Cipriano, L. Gaspero, and A. Dovier, "A hybrid solver for large neighborhood search: Mixing gecode and easylocal++," in *Hybrid Metaheuristics* (M. J. Blesa, C. Blum, L. Gaspero, A. Roli, M. Sampels, and A. Schaerf, eds.), vol. 5818, pp. 141–155, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.