_____

A Large Neighbourhood Search Approach to the Multi-Activity Shift Scheduling Problem

Claude-Guy Quimper
Louis-Martin Rousseau

December 2007

CIRRELT-2007-56

A Large Neighbourhood Search Approach to the Multi-Activity Shift Scheduling Problem

Claude-Guy Quimper[1] , Louis-Martin Rousseau[2,*]

[1] Omega Optimisation, 4200 St-Laurent, suite 301, Montréal, Canada H2W 2R2

[2] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT), and Department of Mathematics and Industrial Engineering, École Polytechnique de Montréal, C.P. 6079, succursale Centre-ville, Montréal, Canada H3C 3A7

Abstract. The challenge in shift scheduling lies in the construction of a set of work shifts, which are subject to specific regulations, in order to cover fluctuating staff demands. This problem becomes harder when multi-skill employees can perform many different activities during the same shift. In this paper, we show how formal languages (such as regular and context-free languages) can be enhanced and used to model the complex regulations of the shift construction problem. From these languages we can derive specialized graph structures that can be searched efficiently. The overall shift scheduling problem can then be solved using a Large Neighborhood Search. These approaches are able to return near optimal solution on traditional single activity problems and they scale well on large instances containing up to 10 activities.

Keywords. Shift scheduling, large neighbourhood search, very large scale neighbourhoods, formal languages.

_____

* Corresponding author: Louis-Martin.Rousseau@cirrelt.ca

1. Introduction

Shift scheduling is described as the problem of constructing a global work schedule so that enough employees are performing tasks at a given time to satisfy the demand. The schedule of an employee is subject to many regulations such as break and meal placement. In some organization, such as contact center or grocery stores for instance, employees have multiple skills and are asked to perform a different activities within one shift. Sequencing these activities in time is again subject to many complex rules.

In this paper, we show how regular and context-free languages can be used to model the regulations dealing with the construction of a shift. General algorithms addressing both formal languages are derived and used in a *Large Neighborhood Search* [34] to derive good solutions to typical shift scheduling problems. The ability to completely schedule one individual (through these large neighborhoods) is a key issue in rapidly producing very good results to complex shift scheduling problems.

Formal langages have been introduce to solve combinatorial optimization problem in the context of Constraint Programming (we refer the reader to [2, 13] for more information on this paradigme). Pesant [27] introduced the REGULAR constraint that forces a sequence of characters to belong to a regular language and Demassey *et al.* [14] extended this constraint to the COST-REGULAR constraint in order to accept only sequences whose cost are below a certain threshold. Both Sellmann [33] and Quimper and Walsh [28] introduced the GRAMMAR constraint that forces a sequence of characters to belong to a context-free language.

The idea of using grammars to model an operator that modifies a sequence of variables has been introduced by Bonaparte and Orlin in [7]. They show how grammars can be used to model most of the previously known neighborhoods of the traveling salesman problem (TSP) and show how a general algorithm, based on these grammars, can be used to search these neighborhoods. What is proposed here is somewhat different since we use automata and grammars to model the scheduling problem itself and search the associated graph to generate the best solution to a complete sequence of variables. This would be impossible with the TSP since one would need an automaton or a grammar of exponential size to capture the Hamiltonian nature of the problem.

The contribution of this approach lies in using the modeling power of these languages, which allow to describe in a formal way the many complex realities of multi-activities shift scheduling problems. Furthermore, the structure induced by the graph representation of these models can be very efficiently searched with either path or tree based algorithms. These structures are thus used to define *Very Large Scale Neighborhoods* [1] which are search spaces exponential in size that can be traversed in polynomial time through the use of specialized algorithms. The motivation to use such a neighbourhood comes from the intuition that larger neighborhoods tend to minimize the probability of getting caught in a local minimum (see for instance [32, 25]).

The structure of the paper is as follows. We first introduce in Section 2 the shift scheduling problem and propose a brief overview of the vast related literature. Section 3 (resp. 4) shows how one can model and solve the individual shift construction problem using automata (resp. context-free grammars). The difference between regular and context-free grammars is discussed in Section 5. Experimental results are presented in Section 6 for

both realistic instances and known benchmarks. We finally conclude and suggest some future works.

2. The Shift Scheduling Problem

This section first describes in detail the shift scheduling problem and introduces the notation that will be used throughout the rest of the paper. It then reviews some important related work, addressing both the single and multiple activity shift scheduling problem.

2.1. Problem Description: the Multi-Activity Context

This shift scheduling model describes a multi-activity work environment in which employees may perform several tasks within each shift and can have limited availability. In this case, the shift scheduling problem is no longer limited to the specification of work periods but must also handles the assignment of activities to employees.

The multi-activity shift scheduling problem consists of determining when and on which activities a set of n employees should work at a given time. The time span to be scheduled is usually divided into T time slots of equal durations. For each time slot, an employee can either work on one of the m activities or rest. A schedule is a set of n sequences of length T denoted by $S = \{s_1, \ldots, s_n\}$. The schedule of an employee e is given by the sequence s_e of T characters. The ith character denoted by $s_e[i]$ represents the state of the employee during the ith time slot. For instance, $s_e[i] = \langle break \rangle$ indicates that the employee is in break while $s_e[i] = \langle activity_1 \rangle$ indicates that the employee is working on the activity number one. The schedule of an employee is subject to some *scheduling rules* defining when an employee can work, be on break, switch activities, etc.

Let $\mathrm{act}(S, t) = \{s_i[t] \mid s_i \in S\}$ be the multiset of activities executed at time t (notice that an activity might occur more than once in a multiset). The supply of activity a in schedule S at time t is given by $\mathrm{occ}(a, \mathrm{act}(S, t))$, i.e. the number of occurrences of activity a in the multiset $\mathrm{act}(S, t)$. The function $d(a, t)$ returns the demand for an activity a at time t. There is an *under cost* of $\underline{c}(a, t)$ for not satisfying the demand (also called opportunity cost) and an *over cost* of $\overline{c}(a, t)$ for over-satisfying the demand (typically computed as excess salary minus additional incomes). Moreover, there is a fixed cost of $c(a, t)$ for asking an employee to work on an activity a at time t. Assuming that $A = \mathrm{act}(S, t)$ is the multiset of activities assigned to the employees at time t, the cost associated with activity a at time t is given by the cost function.

$$\mathrm{cost}(a, A, t) = \mathrm{occ}(a, A)c(a, t) + f(\mathrm{occ}(a, A) - d(a, t), a, t)$$
$$f(x, a, t) = \begin{cases} x\overline{c}(a, t) & \text{if } x \geq 0 \\ -x\underline{c}(a, t) & \text{otherwise} \end{cases}$$

The function $cost(A, t)$ returns the cost for having the multiset of activities A executed at time t.

$$cost(A, t) \; = \; \sum_{a=1}^{m} \mathrm{cost}(a, A, t) \tag{1}$$

The overall cost associated to a schedule S is the sum of the costs associated with each time slot.

$$cost(S) \; = \; \sum_{t=1}^{T} cost(\mathrm{act}(S, t), t) \tag{2}$$

A *satisfiable schedule* is a schedule such that each shift s_i satisfies the scheduling rules. An *optimal schedule* is a satisfiable schedule whose cost is minimal.

2.2. *Literature Review*

Employee Scheduling and Rostering Problems have been extensively studied in the scientific literature, with early papers dating from 1954. For a complete literature review on the subject, the reader is referred to Ersnt *et al.* [18, 17] for a survey on the topic and a detailed annotated bibliography on more than 700 papers (with 64 papers on the more specific Shift Scheduling Problem).

2.3. *Single-Activity Shift Scheduling Problem*

The Shift Scheduling Problem (SSP) was first introduced by Edie [16] in the context of scheduling toll booth operators and the first integer programming model to address it was proposed shortly after by Dantzig [12]. In this set covering formulation, all the possible variables (shifts that could be performed by the employees) are enumerated a priori. The objective is thus to select the set of shifts that covers demand and minimizes the cost. This approach has since remained very popular with more than 130 papers applying it to Employee Scheduling and Rostering [17] and is used in many commercial workforce management systems. Its strength lies in the fact that the complexity involved in assembling the shifts (choosing length, placing breaks, etc.) is hidden in the enumeration phase. It also introduces an integrality gap in the mixed integer formulation. However, when flexibility is introduced in the model through multiple breaks, multiple break types, multiple activities, or simply longer operating hours, the number of possible shifts becomes quite large. A natural solution to this problem is a Branch and Price approach (proposed by Trick et al. [24]) where the shifts that are still manually generated are only introduced in the model when they are necessary.

In an attempt to address this difficulty, Moondra [26], Bechtold and Jacobs [5, 6], Thompson [35], Aykin [3, 4] and Rekik *et al.* [30] proposed and pursued research on implicit formulations. In these models, shift types, specified by starting time and length, are not coupled with break positions at first. For instance, one can independently decide how many employees are going to work from 8am to 4pm and how many employees are going to be

on break at 10am. Specific constraints (namely forward and backward constraints) have been introduced and refined over time to ensu re the existence of a valid schedule. The actual schedule can later be reconstructed with a polynomial-time algorithm. The main advantage of this approach is that the number of decision variables is significantly reduced compared to the covering formulations. However, it is now much harder to express difficult constraints that regard the construction of individual shifts or even cost structures that depend on the global composition of one shift. Furthermore, none of these approaches are able to schedule multiple activities (even if [30] does mention it as future work).

Some problems are often too large to be solved with an exact method. This is the case for the Tour Scheduling problem where one wants to construct the shifts over one week or more. In such cases, heuristics and metaheuristics are appealing methods. We note the use of tabu search [19], simulated Annealing [9], and genetic algorithms [15]. These approaches, which rely on clever neighborhood structures, are able to quickly produce good solutions for large problems.

2.4. *Multiple-Activity Shift Scheduling Problem*

If the literature on single-activity shift scheduling is very rich, problems dealing with multiple activities have however been seldom studied.

Loucks and Jacobs [22] and Ritzmann *et al.* [31] are among the few papers that attempt to address the multi-activity context. Both papers model the tour scheduling problem (shift scheduling over one week) with Boolean assignment variables, specifying the number of employees assigned to a given task at a given hour. Since such approaches yield very large MIP formulations, they both proposed heuristic techniques to construct and improve the solutions. Furthermore, they do not place breaks or meals during the shift, nor do they handle regulations concerning the transition between activities.

Vatri [36] proposed using a column generation with an heuristic approach to build a schedule for air traffic controllers. The problem solved is limited to a sliding time window (of typically a few hours) and breaks are not taking into accounts. Bouchard [8] extends the work of Vatri to include break placement but both dominance and the search for solutions are done heuristically.

Demassey et al. [14] proposed a column generation approach based on constraint programming and regular languages that can solve the multiple activity case presented here. If computing the linear relaxation through column generation is done efficiently for problems containing up to 10 activities, branching to find integer solutions is much more difficult as the approach can only find solutions to some of the smaller instances, when the linear relaxation is also integer.

3. The Shift Scheduling Problem and Regular Languages

The scheduling rules defining what is a valid feasible schedule may vary from one instance to another. It is essential to define a framework that is flexible enough to encode most cases encountered when modeling a shift scheduling problem.
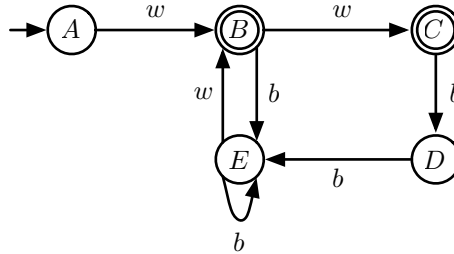
Figure 1. Automaton accepting any shift an employee can work according to the scheduling rules stated in Example 3.1

.

3.1. Modeling the Scheduling Rules

We recall the notion of automaton and show how automata can model the scheduling rules.

An *automaton* is a tuple $A = \langle Q, q_0, F, \Sigma, \delta \rangle$ where Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ are the final states, Σ is an alphabet, and $\delta \subseteq Q \times \Sigma \times Q$ is a transition table. A sequence $S \in \Sigma^n$ is accepted by the automaton A if there exists a sequence of $n+1$ states $r_0, \ldots, r_n \in Q$ such that $\langle r_{i-1}, S[i], r_i \rangle$ is a transition in δ, where $r_0 = q_0$ is the initial state, and $r_n \in F$ is a final state. Automata are often depicted with a directed graph where each state is a node and each transition $\langle q, c, r \rangle \in \delta$ is a directed edge between node q and node r labeled with character c. A language $\mathcal{L}$ over an alphabet Σ is a set of sequences of elements taken from an alphabet Σ. The sequences recognized by an automaton A form a *regular language*. The reader is referred to [20] for more detailed explanations on regular languages and automata.

Lets now look at a automatong that models the scheduling rules as follows.

EXAMPLE: Consider the following scheduling rules.

1. An employee works for a maximum of two consecutive periods (w).

2. Two consecutive working periods are followed by at least two breaks (b).

3. A day may not start nor end on break..

The automaton depicted in Figure 1 accepts all shifts satisfying the three scheduling rules.

□

3.2. VLSN Using Regular Languages

We show how local search can solve the shift scheduling problem when there exists a regular language that fully captures the scheduling rules. Using a simple large neighbourhood
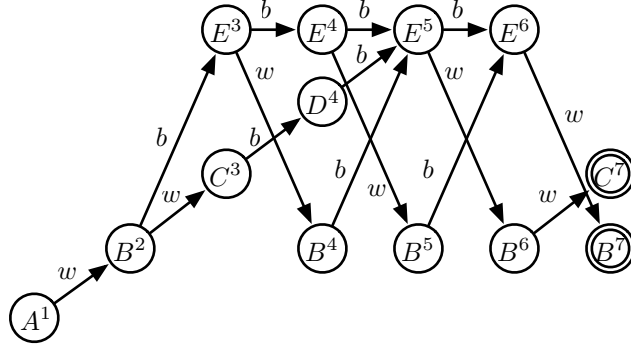
Figure 2. Expanded graph associated with the automaton presented in Figure 1 with $T = 6$.

approach based on dynamic programming, we find a set of work shifts that fits the demand curve and minimizes the overall cost.

Let $\mathcal{A} = \langle Q, q_0, F, \delta \rangle$ be an automaton that recognizes the work shifts satisfying the scheduling rules. We want to find a set of sequences $S = \{s_1, \ldots, s_n\}$ accepted by the automaton such that the overall cost stated in Equation 2 is minimized.

Following [27], we create a layered graph G that we call the *expanded graph*. For every state $q \in Q$ of the automaton, we create $T + 1$ nodes labeled $q^0, \ldots, q^T$. Each node belongs to a different layer of the graph. For every transition $(q, c, r) \in \delta$, we add the edges (q^i, r^{i+1}) to G, for $0 \leq i < T$, that we label with the character c. Finally we remove from the graph G every edge and every node that do not lie on a path connecting the node q_0^0 to one of the node q_f^T where $q_f \in F$ is a final state. Using the automaton of Example 3.1 presented in Figure 1, we obtain the expanded graph of Figure 2.

We define what is a *feasible path*.

Definition 1. [Feasible Path] A feasible path is a path in an expanded graph G that has exactly T edges.

Notice that a feasible path starts with the node q_0^0 and finishes with a node q_f^T where $q_f \in F$ is a final state. Consider any feasible path p. If one follows the edges of this feasible path and prints the characters associated with every edge, we obtain a sequence accepted by the automaton $\mathcal{A}$. In fact, every feasible path in the expanded graph G is associated with a sequence accepted by the automaton and every sequence accepted by the automaton can be retrieved by following a feasible path.

To construct an initial feasible schedule, we randomly select n feasible paths and obtain n sequences $s_1, \ldots, s_n$. We then aim at improving the overall cost of this solution using *Large Neighborhood Search* [34]. To do so, we select a sequence s_r among the n sequences and replace it by a new sequence N that maximizes the improvement of the global solution. The rest of this section discusses how to obtain this new sequence N.

Suppose that at time t, employee r was scheduled to accomplish an activity a but we reschedule this employee to work on activity b i.e. $s_r[t] = a$ and $N[t] = b$. The two activities a and b might be identical or not. With this new assignment, the cost associated with the time slot t can potentially change. Recall that the multiset of activities executed at time t is $\text{act}(S, t)$. The cost of the schedule S at time t is $cost(\text{act}(S, t), t)$. If we replace activity a by activity b, the cost of the new schedule at time t becomes $cost(\text{act}(S, t) \setminus \{a\} \cup \{b\}, t)$.

We assign a cost to every edge in the expanded graph G as follows. Let $e = (a^t, b^{t+1})$ be an edge in G labeled with character c. We assign to the edge e a cost of $cost(\text{act}(S, t) \setminus \{s_r[t]\} \cup \{c\}, t)$. The expanded graph G now has the following property.

LEMMA 1 *Let p be a feasible path and N be the sequence associated with this feasible path. The cost (or length) of p is equal to the cost of the schedule obtained by replacing s_r by N.*

Proof: The tth edge in the path has a cost of $cost(\text{act}(S, t) \setminus \{s_r[t]\} \cup \{N[t]\})$. The sum of the cost of each edge gives $cost(S \setminus \{s_r\} \cup \{N\})$ which is exactly the cost of the schedule obtained by replacing the sequence s_r with N. ∎

It follows from Lemma 1 that the sequence associated with the shortest path in the expanded graph G is the one that better replaces sequence s_r in S. The running time complexity of finding the shortest path in a layered graph is proportional to the number of edges in the graph. In our case, finding the shortest path in the expanded graph G requires $O(T|\delta|)$ steps.

Improving a schedule S can therefore be done by choosing a sequence $s_r \in S$, computing the costs of the edges in the expanded graph according to S and s_r, computing the shortest path in the expanded graph, and finally replacing s_r by the sequence associated with the shortest path. This operation can be repeated until a local minimum is reached, i.e. until the length of the shortest path in the expanded graph G is equal to the cost of the current schedule for any $s_r \in S$. Notice that the graph needs to be computed only once. Only the weights on the edges need to be computed at each iteration.

3.3. Enhancing Regular Languages

One can easily enrich a regular language by modifying the expanded graph. For instance, one could make a state inaccessible for a specific time slot by removing the corresponding node from the appropriate layer. Some transitions could also become unavailable between some layers. Such modifications to the expanded graph are useful to model operations that can only be accomplished at particular times of the day. For instance, if lunches must be taken inside a specific period of the day, one can remove any transition associated with a lunch break outside of that period. Finally, one can set a fixed cost to every transition in the graph. This cost is added to the cost already computed for every edge. Fixed costs are useful to model exceptions in a schedule that involves cost. For instance, employees working after 6pm might require a higher salary.

4. The Shift Scheduling Problem and Context-Free Languages

Context-free grammars are useful since they can encode richer languages than automata. We first formally define what context-free grammars are and then explain how they can encode scheduling rules.

A context-free grammar is a tuple $G = \langle N, S, \Sigma, P \rangle$ where N is a set of symbols called the *non-terminals*, $S \in N$ is the *starting non-terminal*, Σ is a set of symbols called the *terminals*, and P is a set of *productions* of the form $A \rightarrow w$ where $A \in N$ is a non-terminal and w is a sequence of terminals and non-terminals. A *parsing tree* is a tree where the leaves are terminal symbols and the inner-nodes are non-terminal symbols. The root of a parsing tree is the starting non-terminal S. The children of an inner-node A form a sequence w such that the production $A \rightarrow w$ belongs to the grammar. A context-free grammar recognizes a sequence $T \in \Sigma^n$ if there exists a parsing tree whose leaves, when visited from left to right, reproduce the sequence T. We represent non-terminals with capital letters and terminals with lower case letters unless otherwise specified. The abbreviation $A \rightarrow w \mid u$ refers to two distinct productions $A \rightarrow w$ and $A \rightarrow u$.

The sequences recognized by a context-free grammar G form a *context-free language*. A grammar is in the *Chomsky normal form* if every production is either of the form $A \rightarrow BC$ or $A \rightarrow a$, i.e. if a non-terminal produces either two non-terminals or one terminal. Every context-free language can be recognized by a grammar written in the Chomsky normal form. Context-free grammars can recognize any regular language but automata cannot recognize every context-free languages. Context-free grammars therefore encode a richer class of languages.

Context-free grammars can model the scheduling rules as showed in Example 3.1. Notice that the grammar is not in its Chomsky normal form.

$$B \rightarrow b \mid bB \qquad\qquad (3)$$
$$S \rightarrow wBS \mid wwbBS \mid w \mid ww \qquad\qquad (4)$$

4.1. VLSN Using Context-Free Languages

We present a large neighbourhood operator for shift scheduling problems modeled with context-free languages. In the previous section, the certificate proving that a sequence of characters belongs to a regular language was a path in an automaton. We therefore looked for the path in the automaton that induces the smallest cost. The certificate proving that a sequence belongs to a context-free grammar is a parsing tree. We will therefore find the parsing tree that induces the smallest cost to the schedule.

We assume that the grammar modeling a valid schedule for an employee is in its normal Chomsky form such as the grammar introduced in Example 4.1.

EXAMPLE: The grammar G presented be low ensures that an employee executes activity a for an undefined period of time followed by activity b for another undefined period of time.

$$S \rightarrow AB \qquad A \rightarrow AA \mid a \qquad B \rightarrow BB \mid b$$

□

The CYK parser [10, 21, 37] is an algorithm based on dynamic programming that decides whether a sequence belongs to a context-free language. If the sequence belongs to the language, the algorithm returns a parsing tree.

Following [29], we use the CYK parser to build a graph embedding all possible parsing trees (see Figure 3). Algorithm 1 creates a DAG such that the leaf nodes are associated with a terminal character and a position in the sequence. The inner node $N(A, i, j)$ is associated with a non-terminal A, a starting position i, and a span of j. The inner node $N(A, i, j)$ belongs to the graph if there exists a sub-string of length j starting at the ith position that can produce the non-terminal A. If such a node exists, it has children which are a list of pairs of nodes. For instance the node $N(A, i, j)$ could have has children a pair of nodes $\langle N(B, i, k), N(C, i+k, j-k) \rangle$. The relation between the parent and the children indicates that the production $A \to BC \in G$ can produce the non-terminal A at position i with a span of j by concatenating the sub-string of length k starting at position i and the sub-string of length $j - k$ starting at position $i + k$. After building all relations between the nodes, the algorithm removes any node that do not contribute to the production of a parsing-tree, i.e. any node that does not have for ancestor the node $N(S, 1, T)$. Note that $Children(x)$ is the set of all nodes for which x is a direct parent.

1 **for** *all non-terminals A* **do**
2   **for** $i \in [1, T]$ **do**
4   Create node $N(A, i, 1)$
5 $Children(N(A, i, 1)) \leftarrow \{t \mid A \to t \in G\}$

6 **for** $j \in [2, T]$ **do**
7   **for** $i \in [1, T - j + 1]$ **do**
8   **for** *all non-terminals A* **do**
9   Create node $N(A, i, j)$
11 $Children(N(A, i, j)) \leftarrow$
12 $\{\langle N(B, i, k),\ N(C, i+k, j-k) \rangle \mid$
13 $k \in [1, j),\ A \to BC \in G,$
14 $Children(N(B, i, k)) \neq \emptyset,$
16 $Children(N(C, i+k, j-k)) \neq \emptyset\}$

18 Delete any node that does not have the node $N(S, 1, T)$ among its ancestors.

Algorithm 1: This algorithm based on the CYK parser [10, 21, 37] constructs a graph embedding all possible parsing trees of sequences of length T recognized by the grammar G.

Algorithm 1 constructs a graph called the *grammar graph*. Since Algorithm 1 runs in $\Theta(|G|n^3)$ steps, the size of the grammar graph is bounded by $O(n^3 |G|)$. In practice, the grammar graph is usually much smaller as its size is considerably reduced by line 18. The grammar graph obtained from the grammar introduced in Example 4.1 over a sequence of length $T = 3$ is depicted in Figure 3. The grammar graph contains the parsing tree of the
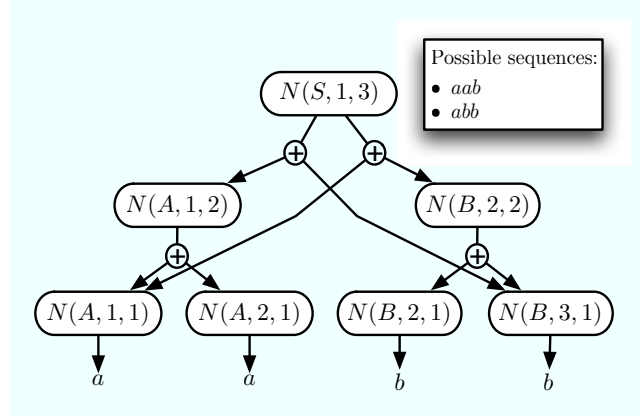
Figure 3. The grammar graph obtained with the grammar introduced in Example 4.1 and $T = 3$. Each plus-node represents a pair of children. For instance, the plus-node bellow $N(A, 1, 2)$ represents the relation $Children(N(A, 1, 2)) = \langle N(A, 1, 1), N(A, 2, 1) \rangle$.

two sequences of length 3 belonging to the language. In fact, any sequence of length T has its parsing tree embedded in the grammar graph as guaranteed by the following lemma.

LEMMA 2 *The grammar graph produced by Algorithm 1 embeds the parsing tree of any sequence of length T accepted by the grammar G.*

Proof: Let S be an arbitrary sequence recognized by the grammar G. There exists a parsing tree certifying that the sequence S belongs to G. The nodes corresponding to the the leaves of this parsing tree are necessarily created on line 3 as the algorithm creates a node for all possible terminals. Every inner node of the parsing tree is connected to its children on line 11. This observation results from the correctness of the CYK parser [10, 21, 37]. Finally, since the root of the parsing tree corresponds to the node $N(S, 1, T)$, no nodes belonging to the parsing tree are removed on line 18. ∎

We can produce a random sequence accepted by the grammar G by traversing the grammar graph from the root and randomly branching on a pair of children until we reach the leaves. Repeating this for every employee, we can produce an initial random schedule $S = \{s_1, \ldots, s_n\}$.

As with regular languages, we improve the schedule S by choosing a sequence $s_r \in S$ that we replace with a new sequence N. This new sequence N minimizes the overall cost of the new schedule. The cost of the time slot t in the schedule S is given by $cost(S[t], t)$. If the activity executed in the work shift N at time t is $N[t] = a$, the new cost for the time slot t becomes $cost(S[t] \setminus \{s_r[t]\} \cup \{a\}, t)$. We will assign weights to the nodes in the grammar graph function of these new costs.

Each leaf of the grammar graph is associated with a particular time slot. A leaf-node $N(A, t, 1)$ is associated with the time slot t. For every leaf-node $N(A, t, 1)$ and for every character a appearing in a production $A \rightarrow a \in G$, Algorithm 2 computes the cost of

the assignment $N[t] = a$. The algorithm selects the character a that induces the smallest cost on the time slot t and assigns a weight equal to this cost to the leaf-node $N(A, t, 1)$. The weight of a parsing tree is equal to the sum of the weights assigned to its leaves. The following relation holds between a sequence N, its parsing tree, and the cost of the schedule $S \setminus \{s_r\} \cup \{N\}$.

LEMMA 3 *Algorithm 2 assigns to node $N(S, 1, T)$ a weight equal to the smallest cost of a schedule obtained by replacing the sequence s_r by a sequence N.*

Proof: Algorithm 2 computes the cost of the parsing tree with the smallest weight. This fact follows from the proof of optimality of the weighted-CYK algorithm [23]. Let N be the sequence associated with this parsing tree. Any parsing tree has exactly one leaf per time slot t with cost $cost(S[t] \setminus \{s_r[t]\} \cup \{N[t]\}, t)$. The sum of the costs of every leaf is $cost(S \setminus \{s_r\} \cup \{N\})$ is exactly the cost of the schedule S where we replaced the individual sequence s_r by N. ∎

To retrieve the sequence N whose cost is computed by Algorithm 2, one simply needs to backtrack in the grammar graph to the nodes that minimize the expressions on lines 3 and 6.

1 **for** *every leaf node $N(A, i, 1)$* **do**
3 $\quad$ weight$(N(A, i, 1)) \leftarrow \min\{cost(S[i] \setminus \{s_r[i]\} \cup \{a\}, i) \mid A \rightarrow a \in G\}$
4 **for** *every inner node $N(A, i, j)$ in post-order* **do**
6 $\quad$ weight$(N(A, i, j)) \leftarrow \min\{$weight$(N(B, x, y)) +$ weight$(N(C, w, z)) \mid$
7 $\quad\quad\quad \langle N(B, x, y), \ N(C, w, z)\rangle \in$
8 $\quad\quad\quad Children(N(A, i, j))\}$
9 **return** weight$(N(S, 1, T))$

Algorithm 2: Compute the cost of the best schedule obtained by replacing $s_r \in S$ by another sequence.

The local search repeatedly chooses a sequence s_r from S and replaces it by the sequence N that minimizes the cost of the overall schedule. This process continues until it reaches a local minimum. Notice that the grammar graph only needs to be constructed once. Only the weights need to be recomputed at each iteration.

4.2. Enhancing Context-Free Languages

Following [29], one can enrich a context-free language by imposing some conditions on the applicability of a production $A \rightarrow BC$. For instance, one can restrict A to represent a sequence of a specific length. The non-terminals B and C might also have to meet some requirements about the length of the sequences they represent. Similarly, a production can be restrained to be effective only at particular times of the day. We denote a constrained literal A with A_S^P where $P \subseteq \mathbb{N}$ is the set of positions where A can occur in the sequence and $S \subseteq \mathbb{N}$ is the set of valid spans for the subsequence A. If the symbol S or P are

omitted, we assume that the set of valid positions and valid spans is $\mathbb{N}$. A weight w can also be associated with a production. Each time a production is used in the schedule of an employee, the weight of the production is added to the weight of the parsing tree. We indicate the weight w of a production on the arrow as follows: $A \xrightarrow{c} BC$. One can omit to mention the weight whenever it is null.

For instance, the production $A_{[2,4]} \xrightarrow{8.9} BC_{[1,3]}^{\{2,5,7\}}$ can be applied only if the sequence obtained has a length between 2 and 4 characters, the span of the literal C is between 1 and 3, and the subsequence C appears at positions 2, 5, or 7. The weight associated with this production is 8.9 and the literal B is unconstrained.

These new restrictions on the productions are useful to model situations where a worker must work a certain amount of time before changing activities or can have a lunch break only at lunch time. Weights on production can be used to encode preferences or to model the case when extra salary must be given to an employee doing some over-time.

Implementing these constraints only require minor modifications to the algorithm. On line 16 in Algorithm 1, when processing the production $A_{S_a}^{P_a} \xrightarrow{w} B_{S_b}^{P_b} C_{S_c}^{P_c}$, we add the extra-conditions that $i \in P_a$, $i \in P_b$, and $i + k \in P_c$ as well as $j \in S_a$, $k \in S_b$, and $j - k \in S_c$.

Each time a production appears in the parsing tree of a sequence, its weight is added to the total weight of the parsing tree. Line 3 in Algorithm 2 can take into account the weight of the production $A \xrightarrow{w} a$ when computing the weight of a leaf. Line 6 could also take into account the weight of production $A \xrightarrow{w} BC$ when computing the combined weight of a pair of children.

5. Regular Languages vs Context-Free Languages

The choice between an automaton or a context-free grammar to model the scheduling rules does not influence the local search since the large neighbourhood operators return the same sequence in either case. However, we observed that the grammar graph generally has fewer nodes and fewer pairs of children than the expanded graph has nodes and edges. Nodes in an expanded graph must carry the information accumulated from the beginning of the sequence to the end. This information can include, for instance, the number of breaks encountered so far in the sequence. In grammar graphs, nodes only need to carry the information about a sub-sequence which is independently of what precedes or succeeds this sub-sequence.

Consider for instance a language that encodes any sequence starting with either an a or a b followed by a sequence of c's and finishing with the same caracter it started with. The automaton and the context-free grammar on Figure 4 both encode this language. The states 2 and 3 of the automaton encode the same pattern than the states 4 and 5. This redundance is necessary to carry the information about the first caracter of the sequence. However, the production $C \rightarrow cC \mid c$ does not carry any information about what precedes or follows the subsequence of c's. In many cases, the duplication of states in an automaton produces expanded graphs larger than grammar graphs. Since the running time of the large neighbourhood operators are proportional to the size of the graph containing all the valid parsings, the context-free grammars often offer lower computational time.
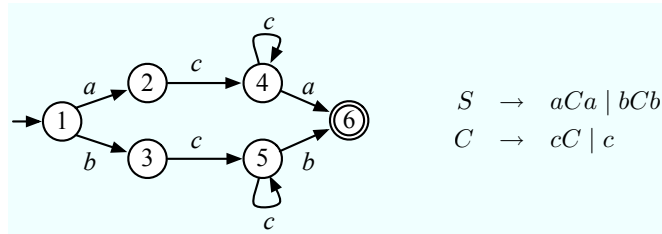
Figure 4. An automaton and an context-free grammar encoding the language $ac^*a \mid bc^*b$. The states 2 and 3 encode the same pattern than the states 4 and 5 while the context-free grammar has only one production to encode the pattern c^*

It is also generally easier to model the scheduling rules with context-free grammars than with automata. Context-free grammars are more expressive. For the scheduling problem solved in our experiments, a context-free grammar of 14 productions captures all the scheduling rules while the equivalent automaton requires thousands of states.

6. Experimental Results

6.1. Problem Description

We consider the following shift scheduling problem with n employees and m activities over a period of time discretized into $T = 96$ time slots of 15 minutes each. During each time slot, an employee can either rest, be in break, lunch, or work on a given activity.
  The schedule of an employee must respect the following rules.

1. A break has a duration of 15 minutes.

2. A lunch has a duration of 1 hour.

3. A part-time employee can take one break during his shift.

4. A full-time employee is entitled to one break, one lunch, and another break in this order.

5. When an employee starts working on an activity, he works for at least one hour on the same activity.

6. Lunches and breaks are scheduled between two periods of work.

7. An employee must have a break or a lunch before changing activities.

8. Periods of rest are to be scheduled at the beginning or the end of the day.

9. A part-time employee must work a minimum of 3 hours and fewer than 6 hours a day.

10. A full-time employee must work a minimum of 6 hours and a maximum of 8 hours a day.

11. At specific times of the day, when the business is closed, every employee must either rest, lunch, or be on break.

As an example, a full time employee could rest between midnight and 9am, then work on activity 1 from 9am to 10:30am, break, work on activity 2 from 10:45am to 12:00pm, lunch, work on activity 3 from 1:00pm to 2:30pm, break, work on activity 4 from 2:45pm to 5:00pm, and rest until midnight.

We model the rules using the following context-free grammar. Identifiers between delimiters $\langle \dots \rangle$ are terminal symbols.

$$R \rightarrow \langle rest \rangle\, R \mid \langle rest \rangle \qquad\qquad A_i \rightarrow \langle activity_i \rangle^{B_i}\, A_i \mid \langle activity_i \rangle^{B_i}$$
$$W_{[4,\infty)} \rightarrow A_i \text{ for } 1 \leq i \leq m \qquad P \rightarrow W\, \langle break \rangle\, W$$
$$F \rightarrow PLP \qquad\qquad S \rightarrow RP_{[17,24]}R \mid RF_{[30,38]}R$$
$$L \rightarrow \langle lunch \rangle\, \langle lunch \rangle\, \langle lunch \rangle\, \langle lunch \rangle$$

Following section 4.2, we constrained some productions in the following way. The production $W_{[4,\infty)} \rightarrow A_i$ can only produce a sequence of at least four characters representing a minimum of one hour of work. In the production $S \rightarrow RF_{[30,38]}R$, F must be a sequence of 30 to 38 characters as a full time employee must work between 6 and 8 hours without counting breaks and lunch. In the production $S \rightarrow RP_{[17,24]}R$, the length of the sequence produced by P must be between 17 and 24 characters long which corresponds to the amount of work time assigned to a part-time employee plus a 15-minutes break. Finally, to model rule 11, we restrict the productions $A_i \rightarrow \langle activity_i \rangle^{B_i}\, A_i\, \langle activity_i \rangle^{B_i}$ to be applied only at positions B_i associated with business hours for activity i.

We tested our model on the first benchmark introduced in [14]. The benchmark has 10 instances with m activities where m ranges from 1 to 10 for a total of 100 instances.

Experimental settings

The implementation was performed in C++ using the compiler gcc 4.1.0 provided in SUSE Linux. The experiments were run on an AMD Opteron 2.3 GHz. The sequence $s_r \in S$ are replaced in a round-robin way and the search is performed for one hour for every instance. Each time the search reached a local optimum, the search was restarted from a random initial solution. We used both the grammar described above and an automaton encoding the same language, but since this automaton contains thousands of states, it was generated by a another C++ program.

Table 1 compares the differences between a local search using an automaton and the same local search using a grammar. The columns are: m the number of activities, $|G_g|/|G_a|$ is the ratio between the size of the grammar graph and the size of the expanded graph, T_g^a and T_a^a (resp T_g^m and T_a^m) are the average (resp median) time, in milliseconds, to evaluate a neighbor using a grammar and an automaton, $\sigma_g/\overline{x}_g$ ($\sigma_a/\overline{x}_a$) is the average standard deviation of the solutions returned by search using a grammar (automaton). To obtain these figures we first compute the standard deviation over all the runs for each single instance

Table 1. Comparison of a one-hour search using an automaton and a search using a grammar.

m	$|G_g|/|G_a|$	T_g^a	T_a^a	T_g^m	T_g^m	$\sigma_a/\overline{x}_a$	$\sigma_a^1/\overline{x}_a^1$	$\sigma_g/\overline{x}_g$	$\sigma_g^1/\overline{x}_g^1$
1	1.22	9	5	9	3	3.49 %	0.84 %	3.22 %	0.69 %
2	0.92	14	13	13	10	6.64 %	2.79 %	6.35 %	2.97 %
3	0.75	33	33	32	33	7.21 %	3.00 %	6.99 %	2.88 %
4	0.64	54	64	50	61	5.57 %	2.20 %	5.51 %	1.96 %
5	0.54	80	94	78	90	3.82 %	1.66 %	3.75 %	1.53 %
6	0.47	112	145	108	140	3.88 %	2.13 %	3.95 %	1.86 %
7	0.41	139	194	135	187	3.58 %	2.06 %	3.41 %	1.69 %
8	0.39	165	251	161	242	3.42 %	2.21 %	3.40 %	1.80 %
9	0.38	173	276	167	265	3.13 %	2.04 %	3.26 %	1.78 %
10	0.35	234	378	227	368	3.11 %	2.29 %	3.05 %	1.88 %

and then average them over the 10 instances. Finally $\sigma_g^1/\overline{x}_g^1$ ($\sigma_a^1/\overline{x}_a^1$) is the average standart deviation when we compare the best solution obtain after 1 second of computation time. To evaluate this we divided the results in 3600 independent sets of 1-second run and compared the best solution of each set.

Analysis

The ratio $|G_g|/|G_a|$ between the size of the grammar graph and the expanded graph shows that the grammar graph grows slower than the expanded graph when the number of activities increases. Indeed, the expanded graph is in average smaller than the grammar graph for instances with one activity but is bigger than the grammar graph for larger instances.

Since both search algorithms have a time complexity which is proportional to the size of the graph they traverse, it is thus normal that the average time to compute a neighbor is larger using the grammar operator than the automaton one. We can also deduce that, given graphs of equivalent size, computing a parsing tree is slightly more time consuming than computing a path, since the advantage in favor of grammars only occurs after three activities. Thus for the more classical instances, with only one activity, the approach using an automaton clearly outperformed the grammar since the graph generated by the grammars is about 22% larger than the expanded graph and the search for neighborhoods takes about twice more time. However, when the number of activities increases, the model using grammars becomes much more interesting as it generates graphs which are about three times smaller and allows to explore the neighborhoods almost 40% faster. Furthermore, the grammar model is generally easier to define than the automaton which needs to be programmed when its size becomes too large.

The local minimums reached by both methods are generally of good quality, as the standard deviation is always below 10%. However, in we allow the algorithm to run for 1 second and keep the best solution obtained within that time limit, then the standard deviation is significantly reduced and never exceeds 3%.

Comparison on single activity problems

To evaluate the effectiveness of our approach on the more typical one-activity shift schedul-

Table 2. Comparison of robustness against a MIP model.

m	#	MIP Cost	MIP Time	lower bound	$t_{5\%}$	$t_{1\%}$	min cost
1	1	**172.67**	1.50	172.67	0.22	**0.22**	**172.67**
1	2	164.58	1029.01	162.94	0.27	**0.67**	**162.94**
1	3	169.44	393.96	168.86	0.24	**0.48**	**169.01**
1	4	133.45	39.89	132.12	0.21	**0.24**	**132.67**
1	5	145.67	10.41	144.6	0.20	**0.20**	**145.11**
1	6	135.06	20.36	134.41	0.32	**0.32**	**134.82**
1	7	150.36	**6.25**	149.42	0.35	22.43	**150.06**
1	8	**148.05**	274.92	147.2	1.26	**13.86**	**148.05**
1	9	**182.54**	15.47	182.54	4.32	**4.32**	**182.54**
1	10	147.63	**5.50**	146.44	0.31	16.26	**147.42**

ing problem, we compare our method against the MIP model described in Côté *et al.* [11], which is an exact approach also relying on regular languages. In [11], the authors introduced a relaxation in Rule 4 by allowing a full-time employee to have both breaks before or after the lunch. In order to allow comparisons and test the robustness of our solution, we also relaxed the problem by adding the following productions to the grammar. Note that this relaxation simplifies the automaton, but it makes the problem more complex as it allows the existence of more possible shifts.

$$Q \rightarrow P \langle break \rangle W \qquad F \rightarrow QLW \mid WLQ$$

The model proposed by [11] was solved using CPLEX 10.0 on a 3.2 GHz Intel Pentium 4 while the local search was run on a AMD Opteron clocked at 2.3GHz. The reader should thus be careful when comparing the computation times. Cplex was configured to stop when the solution reaches an integrality gap of 1 % or after one hour of computation.

In Table 2 where we show the results obtained by the MIP and local search solvers, m is number of activities, # is the instance number, *MIP cost* is the cost of the solution returned by the MIP solver, *MIP time* is the time in seconds to solve the problem, *LB* is a linear relaxation lower bound for the cost, $t_{5\%}$ and $t_{1\%}$ are the time for the local search to find a solution at 5 %, and 1 % of the LB and finally *min cost* is the best solution returned by the local search.

The local search finds a solution at one percent of the optimal within 25 seconds for every instance with one activity. For these instances, the solutions returned are either equivalent or better than those found by the MIP solver.

7. Conclusion and Future Works

Using the fact that regular languages and context-free grammars can model the scheduling rules in a shift scheduling problem, we developed two operators that use regular or context-free languages as large neighbourhood search operators. Using these operators in a simple large neighborhood approach, we were able to solve all benchmark problems in only a few minutes.

In the future, we would like to investigate which sequence $r_c \in S$ is the most appropriate to replace. We currently select the sequences in S in a round-robin way. We would like

to choose a sequence that leads to a *good* local minimum and that directs the search to an unexplored portion of the search space. We would like to combine our operators with some meta-heuristics, which would be able to explore more than one local minimum. Another idea is to implement some kind of long term memory, to allow each run to benefit from what was learned in the preceding ones.

References

1. R.K Ahuja, O. Ergun, J.B. Orlin, and A.P. Punnen. A survey of very large scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.
2. K.R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.
3. T. Aykin. Optimal shift scheduling with multiple break windows. *Management Science*, 42(4):591–602, 1996.
4. T. Aykin. A composite branch and cut algorithm for optimal shift scheduling with multiple breaks and break windows. *Journal of the Operational Research Society*, 49(6):603–615, 1998.
5. S. Bechtold and L. Jacobs. Implicit modeling of flexible break assignment in optimal shift scheduling. *Management Science*, 36(11):1339–1351, 1990.
6. S. Bechtold and L. Jacobs. The equivalence of general set-covering and implicit integer programming formulations for shift scheduling. *Naval Research Logistics*, 43(2):233–249, 1996.
7. A. Bonaparte and J.B. Orlin. Using grammars to generate very large scale neighborhoods for the traveling salesman problem and other sequencing problems. In *Integer Programming and Combinatorial Optimization*, pages 437–451, 2005.
8. M. Bouchard. Optimisation des pauses dans le problme de fabrication des horaires avec quarts de travail. Memoire de maitrise, Ecole Polytechnique de Montreal, 2004.
9. M. Brusco and L. Jacobs. A simulated annealing approach to the solution of flexible labor scheduling problems. *ournal of the Operational Research Society*, 44(12):1991–1200, 1993.
10. J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
11. M.-C. Côté, B. Gendron, and L.-M. Rousseau. Tmodeling the regular constraint with integer programming. In *Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming (CP-AI-OR 07*, pages 29–43, 2007.
12. G. Dantzig. A comment on edies traffic delay at toll booths. *Operations Research 2*, pages 339–341, 1954.
13. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
14. S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.
15. F. Easton and N. Mansour. A distributed genetic algorithm for deterministic and stochastic labor scheduling problems. *European Journal of Operations Research*, 118:505–523, 199.
16. L. Edie. Traffic delays at toll booths. *Journal Operations Research Society of America*, 2(2):107–138, 1954.
17. A. Ernst, H. Jiang, M. Krishnamoorthy, B. Owens, and D. Sier. An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research*, 127:21–144, 2004.
18. A. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153:3–27, 2004.
19. F. Glover and C. McMillan. The general employee scheduling problem: An integration of ms and ai. *Computers and Operations Research*, 13(6):563–573, 1986.
20. J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.
21. T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, 1965.
22. J.S. Loucks and F.R. Jacobs. our scheduling and task assignment of a heterogeneous wok force : a heuristic approach. *Decision Sciences*, 22:719–739, 1991.
23. C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, 1999.
24. A. Mehrotra, K. Murthy, and M. Trick. Optimal shift scheduling: A branch-and-price approach. *Naval Research Logistics*, 47:185–200, 2000.

25. C. Meyers and J.B. Orlin. Very large-scale neighborhood search techniques in timetabling problems. In *The Practice and Theory of Automated Timetabling VI*, pages 36–52, 2006.

26. S. Moondra. An linear programming model for work force scheduling for banks. *Journal of Bank Research*, 6:299–301, 1976.

27. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004*, pages 482–495, 2004.

28. C.-G. Quimper and T. Walsh. Global grammar constraints. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006*, pages 751–755, 2006.

29. C.-G. Quimper and T. Walsh. Decomposing global grammar constraints. In *Principles and Practice of Constraint Programming CP 2007*, pages 590–604, 2007.

30. M. Rekik, J.-F. Cordeau, and F. Soumis. Using benders decomposition to implicitly model tour scheduling. *Annals of Operations Research*, 118:111–133, 2004.

31. L. Ritzman, L.J. Krajewski, and M.J. Showalter. The disaggregation of aggregate manpower plans. *Management Science*, 22:1204–1214, 1976.

32. L.-M. Rousseau, M. Gendreau, and G. Pesant. Using constraint-based operators with variable neighborhood search to solve the vehicle routing problem with time windows. *Journal of Heuristics*, 8(1):43–58, 2001.

33. M. Sellmann. The theory of grammar constraints. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006*, pages 530–544, 2007.

34. P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming CP 1998*, pages 417–431, 1998.

35. G. Thompson. Improved implicit opt imal modelling of the shift scheduling problem. *Management Science*, 41(5):595–607, 1995a.

36. E. Vatri. Integration de la generation de quart de travail et de l'attribution d'activites. Memoire de maitrise, Ecole Polytechnique de Montreal, 2001.

37. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.